# GUARANTEES BY CONSTRUCTION

Types for deadlock and leak free concurrency • separation logics for verified message passing • general and efficient coalgebraic automata minimization • paradox-free probabilistic programming.
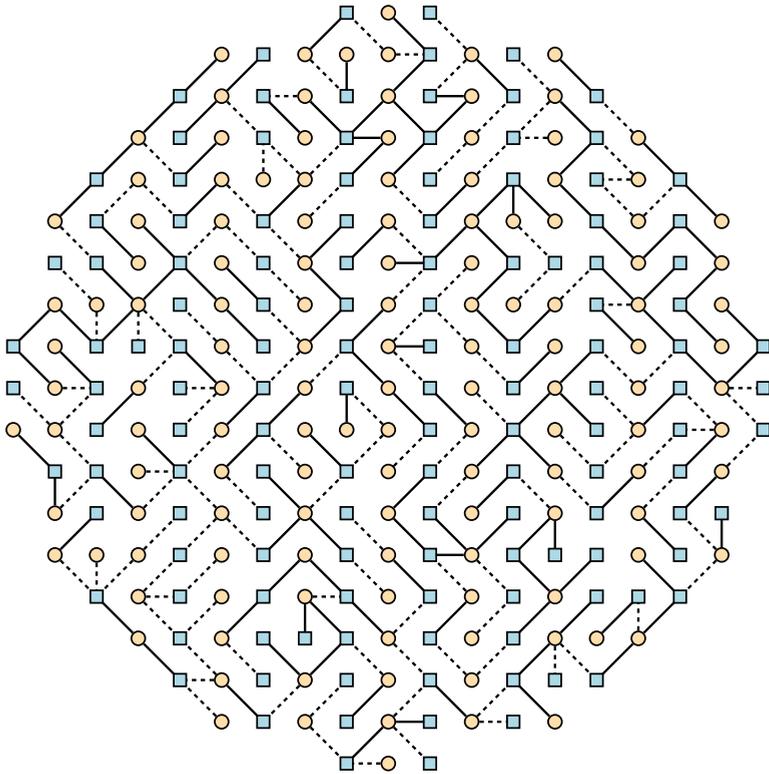
JULES JACOBS

**RADBOUD
UNIVERSITY
PRESS**

INSTITUUT VOOR
ipa
PROGRAMMATUURKUNDE EN ALGORITMIEK

Guarantees by Construction

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnicus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op maandag 24 juni 2024
om 16:30 uur precies

door

Jules Jacobs
geboren op 17 oktober 1989
te Nijmegen

Acknowledgments

First and foremost, I thank Robbert Krebbers and Stephanie Balzer. Without your support, both technical and non-technical, this thesis would not have been possible. I enjoyed my time doing a PhD with you, and I learned a lot from you both. Your teaching of technical expertise and the skill of writing papers was valuable, but perhaps even more valuable was what I learned from the way you work. One particular lesson comes to mind: when starting a new project, it is not necessary nor advisable to first solve the research problem in one's mind. If you just start doing, and sketch, build, write, and discuss, then obstacles that looked like mountains from afar turn out to be molehills when you get up close.

The papers that this thesis consists of were a collaborative effort, not just between Robbert, Stephanie, and myself, but also my other co-authors Jonas Kastberg Hinrichsen and Thorsten Wißmann, who both helped develop the material in this thesis. It was a pleasure to collaborate with you, and I hope to do so again in the future.

I also thank Herman Geuvers, for his encouragement and helping me navigate the academic world; Eelco Visser, for his research that inspired me to start a PhD; Arjen Rouvoet, for helping me take a step back from daily research and think about the bigger picture; Ike Mulder, for the helpful whiteboard discussions and for being a great office mate; and Sriram Sankaranarayanan, for his guidance in support of my first paper. During my PhD, I met many other wonderful people: Dongho Lee, Cas van der Rest, Anton Golov, Jorge Pérez, Bas van den Heuvel, Dan Frumin, Luís Caires, Pedro Rocha, Fabrizio Montesi, Bernardo Toninho, Hans-Peter Deifel, Stefan Milius, Jurriaan Rot, Hubert Garavel, Sebastian Junges, Marc Hermes, Joost-Pieter Katoen, Ahmad Salim Al-Sibahi, Sam Staton, Christian Weilbach, Alex Lew, the research groups from Delft and Nijmegen, and many others. Some of you I met in person, others only online, due to COVID. I thank you all for the interesting discussions, the fun times, the feedback, and the support.

I am also indebted to the anonymous reviewers of my papers and the members of the manuscript committee, for their time and effort in reviewing this thesis. Thanks to Frits Vaandrager, Ralf Jung, Silvia Ghilezan, Steven Holtzen, and Vasco Vasconcelos. I am particularly grateful to Ralf Jung for going above and beyond in providing very detailed feedback and suggestions.

Last but not least, I thank Jo, whose encouragement resulted in me starting a PhD; Firat, for his unwavering support through thick and thin; my mom and dad, for their support and love; Kamiel and Jef, for being awesome brothers; and Noah, for being the cutest nephew. I look forward to seeing you all in the Netherlands soon!

Contents

INTRODUCTION

Introduction

In a world driven by computer systems, the importance of ensuring that these systems function accurately and reliably cannot be overstated. As these systems underpin critical infrastructure, their failure or malfunction can result in catastrophic consequences, both economically and in terms of human well-being. Additionally, with the rise of cyber threats, even a minor vulnerability can be exploited to compromise or disable important systems.

Formal methods aim to improve this situation by providing a rigorous approach to system design and verification based on logic and mathematical reasoning to describe and ascertain the behavior of systems. At their core, formal methods are driven by the pursuit of guarantees that systems behave as expected under all circumstances, thereby providing the confidence and assurance required for their adoption in various critical sectors. Often, these guarantees are manifested in the form of theorems or properties that hold true for an algorithm or a mathematical model of a computer program.

While establishing full correctness theorems for all software would be ideal, this is costly and time-consuming. Moreover, it requires a well-defined and unambiguous per-program specification, which is not always readily available or sufficiently clear, since the desired behavior of software programs may be difficult to quantify with a mathematical formula. Fortunately, many bugs that arise in practice violate general properties that are desirable for all programs. For example, Microsoft and the Chromium team report that 70% of bugs are memory safety violations Thomas (2019); Chromium (2020).

Consequently, the common thread of this thesis is identifying and guaranteeing properties that are generally desirable for all programs and do not require a separate specification for each particular program. We focus on properties that can be expressed with mathematical rigor, but can be guaranteed to hold without much additional effort. We aim to assure these properties inherently via the tool used to create the software system, rendering specific case-by-case validation easy or even unnecessary. In other words, we aspire to ensure beneficial properties *by construction*. Through this approach, we aim to bolster the efficiency, reliability, and overall quality of systems in several areas of computer science.

GUARANTEES BY CONSTRUCTION   We take a *guarantee obtained by construction* to mean a beneficial property that is mathematically assured by the tool used to create the software system and does not require a detailed specification and verification for every new system one creates with the tool. Examples of such properties are abundant in computer science and include the following:

- **Type safety** (Milner, 1978; Wright and Felleisen, 1994; Harper, 2016; Pierce, 2002) in programming languages such as Java, which guarantees that a program will not crash due to performing an illegal operation such as accessing freed memory, or ML, which additionally avoids null pointer exceptions. Type safety means that this property is guaranteed for every type checked program in the language. This does not require a detailed specification of the program's behavior, nor does it require any costly programmer effort to validate, as type checking is done by the compiler.

- **Data race freedom** in Rust (The Rust Team, 2023c), which ensures that programs will not modify and read the same memory location concurrently, which would otherwise be undefined behavior. This property is guaranteed for every well-typed Rust program, as proved by Jung et al. (2018a).

- **Refinement** for verified optimizing C compilers such as CompCert (Leroy, 2006), which formally guarantees that the generated code will behave as specified by the semantics of the source language. This property can be particularly strong for domain specific languages, such as parser generators (Knuth, 1965; Jourdan et al., 2012), which guarantee that the generated parser will correctly parse any input that conforms to the grammar.

- **Time complexity**, such as for regular expressions and LR(k) grammars (Knuth, 1965), which guarantee linear time parsing, or for Datalog (Ceri et al., 1989), which guarantees polynomial time evaluation.

GUIDING PRINCIPLES   Work in this direction is often guided by principles that this thesis also aims to uphold, namely:

- **Abstraction:** the user of the system need not understand its details to use it and benefit from its guarantees. The rules of the system should be simple and easy to understand, even though the internal workings may be complex and the guarantees difficult to prove, *e.g.,* the user of a parser generator need not understand the details of the parsing algorithm to benefit from its correctness and efficiency guarantees.

- **Compositionality:** the properties of compound systems follow from those of their parts, rather than from a global analysis, *e.g.,* a type checker analyses each function in isolation, and the type safety of a compound program follows from the safety of each function, rather than from a global analysis of the entire program.

- **Efficiency:** the system aims to be efficient in terms of its own operation, and in terms of the operation of any further artifacts it may generate, *e.g.,* a type checker should scale well with respect to the size of the program, a compiler should generate efficient code, and a regular expression matcher should run in linear time with respect to the size of the input text.

THIS THESIS   This thesis aims to extend guarantees by construction in four areas: concurrency, separation logic, probabilistic programming, and automata theory.

- For **concurrency**, we aim to extend the guarantees provided by languages like Rust by designing new type systems that guarantee *deadlock and memory leak freedom*.

- For **separation logic** (Reynolds, 2002; O'Hearn and Pym, 1999; O'Hearn et al., 2001; O'Hearn, 2004; Brookes, 2004), we develop a new technique for proving the soundness of separation logics for message passing (Hinrichsen et al., 2020), and we design a new separation logic that guarantees *deadlock and memory leak freedom*.

- In **probabilistic programming** (Goodman et al., 2008; van de Meent et al., 2018), the output of a program may paradoxically depend on the choice of unit (such as meters versus inches). We argue that this is undesirable and aim to change the language so that programs are covariant under general parameter transformations, rendering the language *paradox-free*.

- For **automata theory** (Hopcroft et al., 2007), we aim to provide a general bisimilarity minimization algorithm that can be instantiated for a whole class of automata whose transition structure is specified by a functor, to yield algorithms that compute the *minimal automaton* in *log-linear time complexity* and are *efficient in practice*.

PART 1: TYPES FOR DEADLOCK AND LEAK FREE CONCURRENCY

Systems programming is the practice of writing programs on the lower end of the software stack. These programs need more precise control over memory allocation and representation than provided by higher-level languages that use automatic garbage collection. Languages like C and C++ give the programmer full manual control over memory allocation and representation, but this comes at the cost of safety, as the programmer can easily make mistakes that lead to undefined behavior, such as accessing deallocated memory. This is especially difficult to avoid in concurrent programs, where multiple threads may access and modify the same memory concurrently.

Rust[1] addresses this difficulty with an ownership-based type system (Clarke et al., 1998), which provides semi-automatic memory management by deallocating memory whenever a variable goes out of scope. The Rust compiler is able to statically detect errors in this regard, such as the following:

```
fn foo() -> &i64 {
  let x = vec![1,2,3];
  return &x[0];
}
```

---

[1] We focus on Rust here because it is well-known, but this section applies more generally to languages with ownership-based memory management.

The compiler rejects this function because the backing array of the vector x is deallocated when x goes out of scope, making the returned reference &x[0] invalid.

DEADLOCKS AND MEMORY LEAKS   Rust also enables *fearless concurrent programming* (The Rust Team, 2023a) because its type system guarantees memory safety and the absence of data races for concurrent programs. This is done via carefully designed ownership types for concurrency constructs such as threads, locks, and mesage-passing channels.

Concurrent programming in Rust is not completely fearless, however. Rust programs can still deadlock, as the following example shows:

```
// Deadlock using Mutexes
fn swap(x: &Mutex<u32>, y: &Mutex<u32>)
{
  let mut gx = x.lock();
  let mut gy = y.lock();
  let tmp = *gx;
  *gx = *gy;
  *gy = tmp;
}

let x = Mutex::new(1);
let y = Mutex::new(2);

fork { swap(x,y); }
fork { swap(y,x); }
```

This example uses locks, which are a common synchronization primitive in concurrent programming. In Rust, locks are represented by the Mutex<T> type, which protects a value of type T. The Mutex<T> type has a method lock, which returns a mutable reference to the protected value. The lock is automatically released when the mutable reference goes out of scope.

In the example, we first define the function swap, which takes two locks and swaps the values that they protect. We then create two locks x and y, and spawn two threads that call swap on these locks. The program can deadlock if the two threads acquire the locks in an unfortunate order: the first thread acquires x, and the second thread acquires y. If the first thread then tries to acquire y, it will block until the second thread releases y, but the second thread will never release y because it is blocked on x.

The empirical study of Qin et al. (2020) on bugs in real-world Rust programs found that these bugs occur often in practice. In fact, the authors conjecture that deadlocks are exacerbated in Rust because locks are released implicitly when the lock goes out of scope, which makes it harder for programmers to reason about lock lifetimes:

Even though problems like double locking and conflicting lock orders are common in traditional languages too, Rust's complex lifetime rules combined with its implicit unlock mechanism make it harder for programmers to write blocking bug-free code.

Locks are not the only construct that can lead to deadlock. Rust programs can also deadlock when using mesage-passing channels, as the following example shows:

```
// Deadlock using Channels
fn forward(x: Sender<u32>, y: Receiver<u32>)
{
  let msg = y.receive();
  x.send(msg);
}

let (sx,rx) = Channel::new();
let (sy,ry) = Channel::new();

fork { forward(sx,ry); }
fork { forward(sy,rx); }
```

In Rust, channels are represented by the Sender<T> and Receiver<T> types, which represent the sending and receiving ends of a channel that transmits values of type T. In the example, we first define the function forward, which takes a sender and a receiver and forwards a message from the receiver to the sender. We then create two channels sx and sy, and spawn two threads that call forward on these channels. The program will deadlock because the first thread will block on ry.receive(), and the second thread will block on rx.receive(). Neither thread will ever send a message, so neither thread will ever unblock.

Whereas core Rust programs are guaranteed to be memory leak free, Rust programs that use locks and channels can leak memory.[2] Rust locks can leak memory because they provide shared mutable state that can be used to create cyclic pointer graphs, which lead to leaks when used in combination with non-cycle-aware reference counting, as shown in the following examples:

```
// Memory leak using Arc<Mutex<T>>
enum List {
  Nil, Cons(u32, Arc<Mutex<List>>)
}

let x = Arc::new(Mutex::new(Nil));
*x.lock() = Cons(1,x.clone());
drop(x);
```

This example uses the Arc<Mutex<T>> type, which is a reference counted pointer to a lock that protects a value of type T. Rust allows us to create a cycle of reference counted pointers, which can cause a memory leak, as shown in the example.

---

2 Memory can also be leaked with operations such as mem::forget and Rc::drop (The Rust Team, 2023b).

The following example uses channels to create a memory leak:

```
// Memory leak using Channels
let (sx,rx) = Channel::new();
let (sy,ry) = Channel::new();

sx.send(ry);
sy.send(rx);
drop(sx);
drop(sy);
```

In this example, we create two channels sx and sy and send each channel over the other channel. We then drop both channels, but the channels will not be deallocated because they are still referenced by the other channel.

DEADLOCK AND MEMORY LEAK FREEDOM BY CONSTRUCTION   The goal of the first part of this thesis is to develop type systems for concurrent programming that guarantee that all programs that pass the type checker are deadlock and memory leak free. The type checker should nevertheless be expressive enough to verify many common concurrent idioms. Furthermore, we aim to formally prove that the type system is sound, *i.e.,* that all programs that pass the type checker are indeed deadlock and memory leak free with respect to a mathematical model of the language's semantics.

To do so, we build on prior work. For our purpose, the most prominent area of prior work is *session types* (Honda, 1993; Honda et al., 1998), which are types that describe the communication protocol of a message-passing channel. With changes to the original design, linear session type systems guarantee deadlock freedom and memory leak freedom, even when channels are dynamically created and passed around (Wadler, 2012; Caires and Pfenning, 2010). That is, session types allow us to use channels as first-class values in a programming language, without sacrificing deadlock freedom and memory leak freedom.

The goal of Part 1 of this thesis is to extend this line of work in several ways:

- Chapter 1 introduces a unified method for formally proving the soundness of deadlock and leak free type systems, based on *connectivity graphs*.

- Chapter 2 introduces a type system for *locks* that guarantees deadlock and memory leak freedom.

- Chapter 3 introduces a type system for message passing concurrency between multiple participants (Honda et al., 2008), that guarantees deadlock and memory leak freedom, even when multiparty channels are dynamically created and passed around.

- Chapter 4 introduces a type system for message passing concurrency inspired by session types (Honda, 1993), but distilled to its essence.

FORMAL PROOF   The guarantees that we claim for these type systems are formally proven. By formal proof, I mean mathematical proofs that are carried out in full detail down to basic axioms of mathematics. As a result, such proofs can be checked mechanically using a computer program. Machine-checked proofs require us to be absolutely precise about our definitions, theorems, and proofs. We prefer machine-checked proofs because proofs in previous work have not always been correct or complete, and some theorems presented in the session types literature have in fact turned out to be false (Scalas and Yoshida, 2019).

The results of Part 1 and Part 2 of this thesis are formalized and machine-checked with the Coq proof assistant (Coq Team, 2021), making use of the Iris separation logic framework (Jung et al., 2018b; Krebbers et al., 2017b). Coq ensures that the theorems that pass the proof checker are indeed mathematically valid.

The Coq proof assistant is a computer program that consists of three parts:

1. A language for precisely stating theorems.

2. A language for proving theorems.

3. A checker that verifies that a proof indeed proves the desired theorem, and does not contain mistakes.

In Coq, one starts by stating the desired theorem (the "goal"). One then transforms this goal with commands ("tactics") that correspond to one step of mathematical reasoning. Each tactic replaces a goal with zero or more new goals, which are hopefully simpler than the original goal. The proof is complete when no more goals remain. Coq's tactics and proof checker are designed such that tactics can only be applied in ways that the proof checker ascertains to be correct.

To verify deadlock and leak freedom of the type systems of Part 1, we perform the following steps in Coq:

1. We define the syntax of the programming language.

2. We define the typing rules of the language.

3. We define the operational semantics of the language, which specifies how programs in the language execute.

4. We define what deadlocks and leaks are.

5. We write down a theorem in Coq, stating that for all programs that are valid according to the typing rules, deadlocks and leaks never occur during the execution of such programs.

6. We prove this theorem using Coq's tactics.

These proofs can be quite long, because they must be carried out to the smallest mathematical detail. If these proofs were included as an appendix, then this would

add an additional 780 pages to this thesis. Therefore, I have hosted the Coq proofs on a separate web page `https://apndx.org/thesis/`.

The web page is interlinked with the thesis: gear icons (⚙) next to the definitions and theorems in this PDF link to the corresponding formal proofs on the website, and the formal proofs link back to the thesis.[3] As hyperlinks are not clickable on paper, I have also included an index (page 337) that provides the cross-reference.

We will now discuss each of the chapters of Part 1 in more detail.

Chapter 1: Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic

Based on Jules Jacobs, Stephanie Balzer, Robbert Krebbers, POPL (2022)

Chapter 1 introduces the *connectivity graphs* method for proving the soundness of deadlock and leak free type systems. That is, we aim formally prove that if a program type checks, then its run-time behavior is such that it never deadlocks and never leaks memory. To state this theorem, we need to mathematically define what a program is, what the type checker does, what the run-time behavior of programs is, and what it means to deadlock or leak memory.

Despite the active developments in the mechanization of the meta-theory of binary session types (Thiemann, 2019; Rouvoet et al., 2020; Hinrichsen et al., 2021; Tassarotti et al., 2017; Goto et al., 2016; Ciccone and Padovani, 2020; Castro-Perez et al., 2020; Gay and Vasconcelos, 2010), a mechanized proof of deadlock freedom for binary session types with dynamic thread and channel creation and a dynamically changing communication topology (due to higher-order channels) was still outstanding because of the intricacies of reasoning about graphs in a mechanized setting. While the semantics of global and local types of multiparty session types has recently been mechanized (Castro-Perez et al., 2021), and thus global properties such as deadlock freedom have been shown to hold, the result is confined to a single session without dynamic thread and channel creation and without higher-order channels.

A concurrent program's run-time state consists of a set of threads, and the current state of the heap memory. The proofs of deadlock freedom have to establish that the dependency structure among the threads (or processes) and channels (or locks) remains *acyclic*, even in the presence of dynamic thread spawning and higher-order channels (Carbone and Debois, 2010). In order to reason formally about this dependency structure at a higher level, we introduce *connectivity graphs*.

The key problem that connectivity graphs address is that each of the type systems in Part 1 requires a separate and complex deadlock freedom proof. In particular, deadlock freedom of these systems rests on the acyclicity of the topology of interaction between threads and the synchronization constructs such as locks and channels. This type of reasoning is particularly involved when made fully formal,

---

3 The latter will only work when using a PDF viewer that supports external hyperlinks. The PDF viewers that are built into web browsers usually support this.

as we are then unable to use our human graphical intuition to shortcut proof steps. Connectivity graphs aim to solve this by providing a generic proof methodology for reasoning about deadlock and leak freedom, such that the acyclicity reasoning has to be done only once, and can then be applied to various deadlock free languages.

Connectivity graphs are an abstract representation of the communication topology of concurrently interacting entities, which allows us to encapsulate generic principles for reasoning about deadlock and leak freedom. Connectivity graphs are parametric in their vertices (representing entities like threads and channels) and their edges (representing connections between entities) with labels (representing interaction protocols).

At a high level, the idea of connectivity graphs is to maintain the invariant that the graph is acyclic in a certain strong sense. The acyclicity of the graph then rules out memory that is leaked due to reference cycles, and it rules out deadlocks caused by several threads cyclically waiting for each other.

In order to formalise this intuition, connectivity graphs provide two key tools for proving deadlock and leak freedom:

First, they provide *local graph transformations* that preserve the acyclicity of the connectivity graph. These graph transformations are formulated in separation logic (Reynolds, 2002; O'Hearn et al., 2001), such that resource transfer inside the separation logic corresponds to modification of the graph. Crucially, the linearity of the separation logic guarantees that the acyclicity of the graph is preserved *by construction*. This formally captures the essence of the connection between linearity and acyclicity.

Second, they provide a *waiting induction principle* for acyclic connectivity graphs, which allows us to prove global progress properties from local progress properties. This formally captures the essence of the connection between acyclicity and deadlock and leak freedom.

The connectivity graph proof method is used in Chapters 2 to 4 to prove deadlock and leak freedom of the type systems presented there. The applicability to different languages shows the versatility of the method.

Chapter 2: Higher-Order Leak and Deadlock Free Locks

Based on Jules Jacobs, Stephanie Balzer, POPL (2023)
Recipient of a POPL 2023 Distinguished Paper Award.

Chapter 2 contributes a programming language with a type system for locks that guarantees that all programs are deadlock and leak free.

Balzer et al. (2019)'s Manifest Sharing provides a deadlock free type system for locks that carefully controls ordering. Rocha and Caires (2021)'s PaT language provides another deadlock free type system for locks that does not require ordering constraints, and instead restricts the types of data that can be stored in locks. Inspired by this work, we design a new language and type system for deadlock-free locks.

Locks in this type system are *higher-order*, meaning that programs can create and pass around locks as first-class data, and locks themselves can be part of the concurrently shared data that is protected by another lock. This makes the type system expressive enough to verify many common concurrent idioms, including fork/join, promises, shared mutable data structures, and the implementation of session-typed channels in terms of locks.

Despite supporting all these features, all programs are guaranteed to be deadlock free by type checking. Our type system does not impose any additional proof obligations on the programmer, such as lock orders, the traditional approach to guarantee deadlock freedom (Dijkstra, 1965). Instead, deadlock and leak freedom are guaranteed purely by an ownership type system.

Nevertheless, our language is in some respects *more* expressive than lock orders; there are programs that *can* be shown to be deadlock free with our type system, but *cannot* be shown to be deadlock free with a lock order (*i.e.,* deadlock-free programs for which no lock order exists).

The converse is also true: there are programs for which a lock order exists, but cannot be shown to be deadlock free with our type system. We therefore also develop an extension that incorporates lock orders in our type system, combining the strengths of both approaches in a natural manner. This makes the type system more expressive, at the cost of some complexity in the typing rules. Our simpler type system is embedded as the special case when all lock orders are singletons.

Chapter 3: Multiparty GV: Functional Multiparty Session Types With Certified Deadlock Freedom

Based on Jules Jacobs, Stephanie Balzer, Robbert Krebbers, ICFP (2022)

Chapter 3 combines the benefits of linear binary session types (Wadler, 2012) and multiparty session types (Honda et al., 2008), which allow communication between more than two participants.

Session type systems (Honda, 1993; Honda et al., 1998) allow type checking of programs that involve message-passing concurrency. Session types are protocols, which can be seen as sequences of send (**!**) and receive (**?**) actions. They are associated with channels, and specify in what order messages of what type should be transferred. For example, the session type **!Z.?B.end** is given to a channel over which an integer should be sent, after which a boolean is received. More complex session types can be formed with operators for choice (⊕,&), recursion (μ), *etc.*

Aside from ensuring type safety, linear session type systems (Caires and Pfenning, 2010; Wadler, 2012) can ensure deadlock freedom. That means that well-typed programs cannot end up in a state where all threads are waiting to receive a message from another. Deadlock freedom has been extended to a large variety of session type systems (Carbone and Debois, 2010; Fowler et al., 2021; Toninho et al., 2013; Toninho, 2015; Caires et al., 2013; Pérez et al., 2014; Lindley and Morris, 2015, 2016a, 2017;

Fowler et al., 2019; Das et al., 2018). The elegance of these session type systems is that they give deadlock freedom essentially "for free" — the programmer obtains the deadlock freedom guarantee from type checking alone.[4] Moreover, session types are compositional—once functions have been type checked, they can be composed by merely establishing that the types agree. A final strength of session types is that deadlock freedom is maintained in a higher-order setting where session types are embedded in a functional programming language, where closures and channels can be transferred as first-class data over channels.

Session types have further been extended to multiparty session types (Honda et al., 2008, 2016). Whereas binary session types provide channels that allow for communication between two participants, multiparty session types allow for multiparty channels that enable direct interaction between more than two participants. The actions in multiparty global session types are annotated with the participants that the communication happens between:

$$[0 \rightarrow 1]\mathbf{Z}.[1 \rightarrow 2]\mathbf{B}.\mathsf{End}$$

This session type says that party 0 sends an integer to party 1, and then party 1 sends a boolean to party 2. Multiparty session types guarantee deadlock freedom via a consistency check on the global session type.

It may sound like multiparty session types are strictly more expressive than binary session types, but this is not the case. Whereas multiparty session types do guarantee deadlock freedom for one session, multiparty session types typically do not guarantee deadlock freedom when more than one multiparty channel is involved, unless restrictions are placed on the interleaving of messages (Coppo et al., 2013; Bettini et al., 2008; Coppo et al., 2016). This is in constrast to binary session types, where deadlock freedom is guaranteed even when channels are dynamically created and passed around and used without restrictions, as in modern binary session-typed functional programming languages like GV (Wadler, 2012). In short, multiparty session types do not possess all of the aforementioned benefits of binary/linear session types when embedded in a functional language. Therefore, the world of session types is bifurcated into two incompatible worlds. The aim of this chapter is to unify these worlds and combine the unique benefits of both.

We thus develop a new type system where (1) channels support n-to-n communication with multiparty session types, and (2) channels are first-class values in a functional language. Crucially, the type system guarantees leak and deadlock freedom for all programs, when *multiple* multiparty channels are dynamically created and passed around as first-class data. This includes uses that are *higher-order*, in the sense that channels can be sent as messages over other channels, and captured in closures that are themselves passed around as first-class data. This is the first type

---

4 Note that these languages tie channel creation to thread creation in order to offer this guarantee. This is the reason that deadlock freedom is not guaranteed in earlier session type systems such as the original Honda-style session types (Honda, 1993; Honda et al., 1998) or in the original GV (Gay and Vasconcelos, 2010).

system that combines the expressiveness of the binary session types of the functional language GV (Wadler, 2012), with multiparty session types, while providing these guarantees.

Chapter 4: A Self-Dual Distillation of Session Types

Based on Jules Jacobs, ECOOP (2022)

Chapter 4 distills binary session types to their essence by casting them as a minimal extension of linear λ-calculus with concurrent communication.

The session types community has looked for minimal systems that still capture the essence of session types. These minimal systems typically decompose multi-step session types into single-shot types. Kobayashi (2002b); Dardha et al. (2012, 2017) give an encoding of session types into ordinary π-calculus types, and *minimal session types* by Arslanagic et al. (2019) decompose multi-step session types into single-step session types in a π-calculus. Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries (Scalas and Yoshida, 2016a; Padovani, 2017; Kokke and Dardha, 2021a).

This chapter provides a minimal version of session types in the context of lambda calculus. Our extension of lambda calculus adds only a *single* new **fork** construct for spawning threads, and adds no new operations other than **fork**, no new type formers, and no explicit definition of session type duality. Instead, the language uses the linear function function type former $\tau_1 \multimap \tau_2$ for communication between threads, which is what we call *self-dual* to $\tau_2 \multimap \tau_1$.

We give **fork** the following type:

$$\textbf{fork} : ((\tau_1 \multimap \tau_2) \multimap 1) \multimap (\tau_2 \multimap \tau_1) \tag{1}$$

where 1 is the unit type. That is, **fork** gets passed a closure that consumes a function of type $\tau_1 \multimap \tau_2$ and **fork** returns a function of type $\tau_2 \multimap \tau_1$. When both functions are called, the values of type τ_1, τ_2 are exchanged.

Binary session types and their channel operations can be encoded in this language in a type-preserving way. Because of the language's minimality, the deadlock and leak freedom proof is simplified considerably, relative to traditional session types.

PART 2: SEPARATION LOGICS FOR VERIFIED MESSAGE PASSING

In Part 2, we go beyond ensuring that our programs do not exhibit certain invalid behaviors (like crashing, deadlocking, or leaking memory), and aim for *functional correctness*, that is, the stronger property that our programs produce the correct answer. In order to achieve this, we need to go beyond type systems (where checking is done automatically by the type checker), and switch to *program logics*, in which some amount of manual proof work is required. We thus trade automation for

obtaining stronger results. In particular, this part of the thesis builds on *concurrent separation logic* (O'Hearn, 2004; Brookes, 2004), which is a particularly advanced form of Hoare logic (Hoare, 1969) that is suitable for reasoning about concurrent programs that use pointers and mutable state.

We focus on the development of program logics for the verification of message passing programs. This is challenging, because the fact that our program produces the correct answer may rely on the correct behavior of several threads that run concurrently and communicate via mesage-passing channels. Bocchi et al. (2010); Francalanza et al. (2011); Lozes and Villard (2012); Craciun et al. (2015); Oortwijn et al. (2016); Hinrichsen et al. (2020, 2022) have developed program logics that incorporate concepts from session types to verify increasingly sophisticated programs with message-passing concurrency. The protocols of these program logics make it possible to put logical conditions on the messages, allowing one to specify the contents (*e.g.,* the message is an even number) instead of just the shape (*e.g.,* it is an integer). The state of the art is the Actris logic and its descendants (Hinrichsen et al., 2020, 2022; Jacobs et al., 2023). Actris builds on the Iris separation logic framework (Jung et al., 2015, 2016; Krebbers et al., 2017a; Jung et al., 2018b), which is a program logic that allows for advanced reasoning about concurrent programs with shared mutable state, and is mechanized in Coq (Coq Team, 2021). Actris combines the Iris separation logic with *dependent session protocols*, which describe the communication protocol of a message-passing channel. Session protocols are a generalization of session types, which allows the protocol state to depend on the values that are sent over the channel, and allows the transfer of separation logic resources along with the messages. This enables the Actris logic to reason about programs that use message passing in combination with mutable state and locks.

Chapter 5: Dependent Session Protocols in Separation Logic from First Principles

Based on Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers, ICFP (2023)

Chapter 5 present MiniActris, an elegant and simple account of dependent session protocols in concurrent separation logic for a functional language with message-passing. MiniActris aims to retain most features of Actris while being simpler and easier to understand. Actris' soundness proof required Iris invariants as well as custom use of advanced Iris mechanisms such as recursive domain equations (America and Rutten, 1989; Birkedal et al., 2010) and higher-order ghost state (Jung et al., 2016). MiniActris, by contrast, *only* requires Iris invariants to verify that our one-shot channels satisfy one-shot protocols, without any additional need for custom use of advanced features. Inspired by minimalistic session calculi (Dardha et al., 2012), and building on Chapter 4, we subsequently use one-shot channels as a black box on top of which we can define channels with (dependent) session protocols.

MiniActris retains all the features of Actris 1.0 (Hinrichsen et al., 2020), and the variance subprotocols of Actris 2.0 (Hinrichsen et al., 2021), but does not support

asynchronous subprotocols nor the Actris 2.0 ghost theory, as these features appear to be incompatible with MiniActris' design. By giving up on these features, we obtain a number of advantages in terms of simplicity, elegance, and flexibility: support for subprotocols and guarded recursion automatically transfers from the one-shot protocols to the dependent session protocols, and we easily obtain various forms of channel closing. Because the metatheory of our results is so simple, we are able to give all definitions as part of this chapter, and mechanize all our results using the Iris framework in less than 1000 lines of Coq.

An important part of the mechanization is that we obtain Iris' *adequacy theorem* (Jung et al., 2018b, §6.4), which states that if we prove a Hoare triple for our program using the rules of our logic, and if the precondition holds, then the run-time behavior of the program is such that the program never crashes, and the postcondition holds after the program terminates.

Chapter 6: Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing

Based on Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers, POPL (2024)

Chapter 6 introduces a linear concurrent separation logic, called LinearActris, designed to guarantee deadlock and leak freedom for message-passing concurrency. Much like Actris, LinearActris combines the strengths of session types and concurrent separation logic, allowing for the verification of challenging higher-order programs with mutable state through dependent protocols. The key challenge is to prove the adequacy theorem of LinearActris, which says that the logic indeed gives deadlock and leak freedom "for free" from linearity. That is, the crucial difference between LinearActris and Actris is that proving a Hoare triple in LinearActris guarantees not only that the program does not crash, but also that it does not deadlock or leak memory. We prove this theorem by defining a step-indexed model (Appel and McAllester, 2001; Ahmed, 2004; Birkedal et al., 2011; Dreyer et al., 2011) of separation logic, based on *connectivity graphs* (Chapter 1). Step-indexing allows for recursive protocols, as well as term-level recursion without necessitating termination proofs.

To demonstrate the expressive power of LinearActris, we prove soundness of a higher-order (GV-style) session type system using the logical approach to logical relations (Appel et al., 2007; Dreyer et al., 2011; Jung et al., 2018a; Timany et al., 2022). This shows that our logic is strong enough to verify the deadlock freedom of all programs that can be verified by GV-style session types.

Probabilistic programming languages (Goodman and Stuhlmüller, 2014) allow us to write statistical and machine learning models as computer programs that generate samples from the model. Some examples of probabilistic programming languages

are Church (Goodman et al., 2008), Stan (Carpenter et al., 2017), Anglican (van de Meent et al., 2018), Pyro (Bingham et al., 2018), Hakaru (Phan et al., 2019), and Dice (Holtzen et al., 2020).

Probabilistic programs can model conditional probability distributions using *observe statements*. These programs can be run by accumulating likelihood at each observe statement, and using the likelihood to steer random choices and weigh results with inference algorithms such as importance sampling or MCMC.

Chapter 7: Paradoxes of Probabilistic Programming

Based on Jules Jacobs, POPL (2021)

Chapter 7 is about paradoxes that can occur when using observe statements to condition on measure-zero events, that is, events that can happen, but only with probability zero. We argue that naive likelihood accumulation does not give desirable semantics for such observe statements, particularly when the observe statement is executed conditionally on random data.

These paradoxes lead to situations where the answer computed by a probabilistic program depends on the units of measurement used in the program. That is, the answer computed by a probabilistic program can depend on whether we measure length in inches or in meters. This is undesirable because the answer should not depend on the units of measurement used in the program.

Consider the following example probabilistic program:

```
h = rand(Normal(1.7, 0.5))
if rand(Bernoulli(0.5))
   observe(Normal(h, 0.1), 2.0)
end
```

We first sample a value h (say, a person's height) from a prior normally distributed around 1.7 meters and then with probability 0.5 we observe a measurement normally distributed around the height to be 2.0. Each time the program is run, it samples a random value for h under the given observation. When averaging 10000 samples each, a typical output resulting from the standard semantics looks like 1.812 1.814 1.823 1.813 1.806. Suppose that we had measured the height in centimeters instead of meters:

```
h = rand(Normal(170, 50))
if rand(Bernoulli(0.5))
   observe(Normal(h, 10), 200)
end
```

We might naively expect this program to produce roughly the same output as the previous program, but multiplied by a factor of 100 to account for the conversion

of meters to centimeters. Instead, a typical output resulting from the standard semantics looks like 170.1 170.4 171.5 170.2 169.4, which is clearly inconsistent with the earlier result.

The aim of this chapter is to extend probabilistic programming languages with a semantics for observe statements that is unit-consistent and, more generally, covariant under parameter transformations.

The key idea is to model measure-zero events as a limit of positive measure events, and to accumulate *infinitesimal probabilities* rather than probability densities. This resolves the paradoxes, and allows us to give a semantics for observe statements that is unit-consistent. More generally, all programs that one can write in our language are covariant under parameter transformations by construction.

PART 4: GENERAL AND EFFICIENT AUTOMATON MINIMIZATION

The first automata minimization algorithm was by developed by Moore (1956). This algorithm takes as input a deterministic finite automaton, and computes which states are equivalent (*i.e.,* which states accept the same language). Moore's algorithm runs in $O(n^2)$ time, where n is the number of states. Hopcroft (1971) developed a $O(n \log n)$ algorithm.

In Figure 1, three different types of automata are displayed: deterministic finite automata, transition systems, and Markov chains. Each of these types of automata comes with a notion of state equivalence, which tells us which states are behaviourally equivalent. When two states are behaviourally equivalent, they can be merged into one state, leading to a smaller automaton that is equivalent to the original.

Minimization algorithms were developed for these and other automata types such as transition systems (without action labels) (Kanellakis and Smolka, 1983, 1990), labelled transition systems (Valmari, 2009), Markov chains (Valmari and Franceschinis, 2010), Markov decision processes (Baier et al., 2000; Groote et al., 2018), and weighted tree automata (Björklund et al., 2009, 2007).

Recently, these algorithms and system equivalences have been unified using a coalgebraic generalization (Dorsch et al., 2017; Deifel et al., 2019; Wißmann et al., 2021). This coalgebraic algorithm is parameterized by a functor that describes the automaton type. Once instantiated, the algorithm can minimize automata of the given type. This is useful, because there are many different automaton types.

Chapter 8: Fast Coalgebraic Bisimilarity Minimization

Based on Jules Jacobs, Thorsten Wißmann, POPL (2023)

Chapter 8 introduces an automata minimization algorithm that is more general, efficient, and able to handle large systems.

DFA	Transition system	Markov chain
2 ≡ 3, 4 ≡ 5	**1 ≡ 2, 3 ≡ 4**	**2 ≡ 3 ≡ 5**
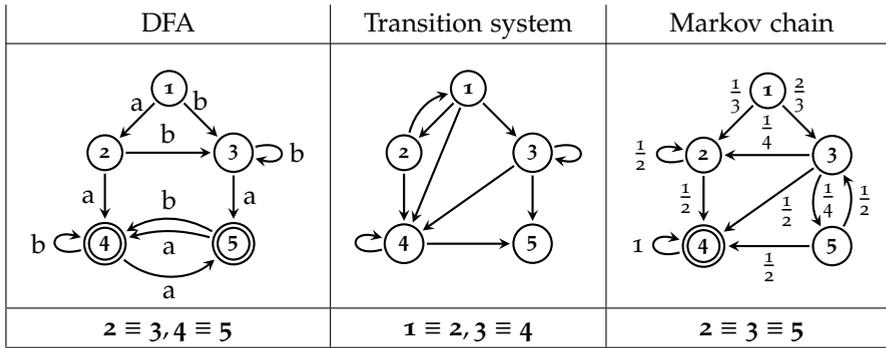
Figure 1: Examples of different automaton types and which states are equivalent.

In particular, we present a generic algorithm that minimizes coalgebras over an *arbitrary functor in the category of sets* as long as the action on morphisms is sufficiently computable. The algorithm makes at most $O(m \log n)$ calls to the functor-specific action, where n is the number of states and m is the number of transitions in the coalgebra.

While more specialized algorithms can be asymptotically faster than our algorithm (usually by a factor of $O(\frac{m}{n})$), our algorithm is especially well suited to efficient implementation, and our tool *Boa* often uses much less time and memory on existing benchmarks and can handle larger automata, despite being more generic.

In summary, we provide a generic bisimilarity minimization method that, when instantiated with a functor that describes the automaton type, guarantees that a minimal automaton is found after at most $O(m \log n)$ calls to the functor.

CONTRIBUTIONS TO THE SCIENTIFIC LITERATURE

1. **Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing** (Chapter 6)
   Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers, POPL (2024)

2. **Dependent Session Protocols in Separation Logic from First Principles** (Chapter 5)
   Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers, ICFP (2023)

3. **Higher-Order Leak and Deadlock Free Locks** (Chapter 2)
   Jules Jacobs, Stephanie Balzer, POPL (2023)
   (Recipient of the POPL Distinguished Paper Award)

4. **Fast Coalgebraic Bisimilarity Minimization** (Chapter 8)
   Jules Jacobs, Thorsten Wißmann, POPL (2023)

5. **Multiparty GV: Functional Multiparty Session Types With Certified Deadlock Freedom** (Chapter 3)
   Jules Jacobs, Stephanie Balzer, Robbert Krebbers, ICFP (2022)

6. **A Self-Dual Distillation of Session Types** (Chapter 4)
   Jules Jacobs, ECOOP (2022)

7. **Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic** (Chapter 1)
   Jules Jacobs, Stephanie Balzer, Robbert Krebbers, POPL (2022)

8. **Long-term ovarian hormone deprivation alters functional connectivity, brain neurochemical profile and white matter integrity in the Tg2576 amyloid mouse model of Alzheimer's disease** (Not included in this thesis)
   Firat Kara, Michael E. Belloy, Rick Voncken, Zahra Sarwari, Yadav Garima, Cynthia Anckaerts, An Langbeen, Valerie Leysen, Disha Shah, Jules Jacobs, Julie Hamaide, Peter Bols, Johan Van Audekerke, Jasmijn Daans, Caroline Guglielmetti, Kejal Kantarci, Vincent Prevot, Steffen Roßner, Peter Ponsaerts, Annemie Van der Linden, Marleen Verhoye, Neurobiology of Aging (2021)

9. **Paradoxes of Probabilistic Programming** (Chapter 7)
   Jules Jacobs, POPL (2021)

CONTRIBUTION STATEMENT The chapters of this thesis are revised versions of peer-reviewed papers. These papers were co-authored with Stephanie Balzer, Robbert Krebbers, Thorsten Wißmann, Jonas Kastberg Hinrichsen, and Firat Kara et al. My own contribution to the papers this thesis is based on is as follows:

- For the four articles on deadlock and leak free type systems coauthored with Stephanie Balzer and Robbert Krebbers (chapters 1, 2, and 3), I was responsible for the key ideas, almost all of the mechanization, and most of the writing. For chapter 4, I was the sole author.

- For the work on message passing in separation logic coauthored with Jonas Kastberg Hinrichsen and Robbert Krebbers (chapters 5 and 6), I was responsible for key ideas, key proofs, and a large part of the writing.

- For the work on probabilistic programming (chapter 7), I was the sole author.

- For the work on coalgebraic bisimilarity minimization coauthored with Thorsten Wißmann (chapter 8), I was responsible for the central ideas of the algorithm and data structures, the implementation and experiments, and a large part of the writing.

- For the work on Alzheimer's disease (not included in this thesis) coauthored with Firat Kara et al., I was responsible for statistical data analysis.

Part I

TYPES FOR DEADLOCK AND LEAK FREE
CONCURRENCY

Chapter 1

Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic

ABSTRACT We introduce the notion of a *connectivity graph*—an abstract representation of the topology of concurrently interacting entities, which allows us to encapsulate generic principles of reasoning about *deadlock freedom*. Connectivity graphs are *parametric* in their vertices (representing entities like threads and channels) and their edges (representing references between entities) with labels (representing interaction protocols). We prove deadlock and memory leak freedom in the style of progress and preservation and use *separation logic* as a meta theoretic tool to treat connectivity graph edges and labels substructurally. To prove preservation locally, we distill generic separation logic rules for *local graph transformations* that preserve acyclicity of the connectivity graph. To prove global progress locally, we introduce a *waiting induction* principle for acyclic connectivity graphs. We mechanize our results in Coq, and instantiate our method with a higher-order binary session-typed language to obtain the first mechanized proof of deadlock and leak freedom.

1.1 INTRODUCTION

Binary session types (Honda, 1993; Honda et al., 1998) are a type discipline for specifying protocols of interactions in message-passing concurrent programs. Session types have turned into an active area of research that enjoys strong theoretical and practical foundations. The theoretical foundations include a Curry-Howard correspondence between session-typed π-calculi and linear logic (Caires and Pfenning, 2010; Wadler, 2012; Caires et al., 2013; Pérez et al., 2014; Toninho et al., 2013; Lindley and Morris, 2015; Toninho, 2015) and session-typed λ-calculi with mainstream programming language features (Lindley and Morris, 2016c, 2017; Igarashi et al., 2017; Fowler et al., 2019). The practical foundations include libraries for session types in mainstream programming languages (Dezani-Ciancaglini et al., 2006; Pucella and Tov, 2008; Imai et al., 2010; Jespersen et al., 2015a; Lindley and Morris, 2016b; Scalas and Yoshida, 2016b; Padovani, 2017; Imai et al., 2019; Kokke, 2019; Chen et al., 2022).

Session-typed languages come with strong guarantees: they not only enjoy *type safety* (a.k.a. session fidelity) but all well-typed programs also enjoy *deadlock freedom* (and consequently, *global progress*). The proofs of deadlock freedom have to establish that the dependency structure among the threads (or processes) and channels (or buffers) remains *acyclic*, even in the presence of dynamic thread spawning and

higher-order channels (Carbone and Debois, 2010). Despite the active developments in the mechanization of the meta-theory of binary session types (Thiemann, 2019; Rouvoet et al., 2020; Hinrichsen et al., 2021; Tassarotti et al., 2017; Goto et al., 2016; Ciccone and Padovani, 2020; Castro-Perez et al., 2020; Gay and Vasconcelos, 2010), a mechanized proof of deadlock freedom for binary session types with dynamic thread and channel creation and a dynamically changing communication topology (due to higher-order channels) is still outstanding because of the intricacies of reasoning about graphs in a mechanized setting. While the semantics of global and local types of multiparty session types has recently been mechanized (Castro-Perez et al., 2021), and thus global properties such as deadlock freedom shown to hold, the result is confined to a single session without dynamic thread and channel creation and without higher-order channels.

In this chapter we develop a parametric proof method for deadlock freedom of concurrently computing entities that interact via shared resources on a dynamically changing acyclic communication topology. We mechanize the proof method in the Coq proof assistant, and instantiate it for a deadlock freedom proof for a variant of GV (Wadler, 2012; Lindley and Morris, 2015), a functional language with higher-order binary linear session types. Proof mechanization has the obvious benefit of providing the peace of mind of a machine-checked proof. Another—maybe even more important—benefit of mechanization is that it encourages us to develop abstractions that encapsulate the reasoning about the acyclic dependency structure of threads and channels, and that shield us from the intricacies of a language's operational semantics and type system.

The key ingredients that make our proof method parametric are our new notion of a *connectivity graph*, to abstract over the dependency structure, and our meta theoretic use of *separation logic* (O'Hearn et al., 2001), to link our abstract connectivity graph to the concrete language's operational semantics and type system. A connectivity graph abstracts concurrent entities and shared resources as vertices and their possible interactions as edges, which are labeled with protocol state. When instantiating the connectivity graph for session types, threads and channels become vertices, channel references become edges whose labels indicate the session type of the referenced channel. By asserting *acyclicity* of the connectivity graph, circular dependencies among the concurrent entities and shared resources are rendered impossible. This guarantees that at any moment at least one interaction can happen (deadlock freedom) and that all channels are deallocated when the program terminates (memory leak freedom).

EXAMPLE   Before we explain the parametric aspects of our proof method, let us consider an example program to see connectivity graphs for linear session types in action:

After both forks: **After T2's send:** **After T1's receive:**
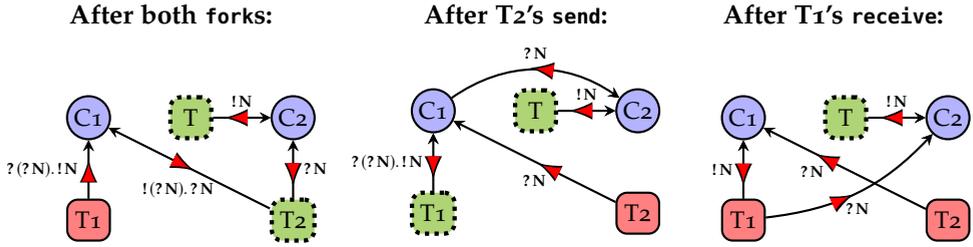
Figure 2: Connectivity graphs with run-time information for our example program (the End markers have been elided from session types). Boxes depict threads (red boxes are blocked threads, green dotted boxes are running threads). Blue cicles depict channels. Black edges indicate references to channel endpoints, labeled with their session type. Red triangles reveal the waiting dependency for each reference to a channel endpoint: either the owner of the endpoint is waiting to receive a message from the channel, or the channel is waiting for the owner of the endpoint to initiate the next action (send or receive or close).

```
1   let c1 = fork (λ c1',        // c1': ?(?N.End).!N.End
2     let (c1',c) = receive(c1') // c1': !N.End, c: ?N.End
3     let (c,n) = receive(c); ...) // c1': !N.End, c: End
4                          T1
5   let c2 = fork (λ c2',        // c1: !(?N.End).?N.End, c2': ?N.End
6     let c1 = send(c1,c2');     // c1: ?N.End
7     let (c1,m) = receive(c1); ...) // c1: End
8                          T2
9                               // c2: !N.End
10  let c2 = send(c2,10); ...    // c2: End
                            T
```

The main thread (T) uses the `fork` construct to spawn two threads (T1 and T2) with bidirectional channels (C1 and C2) connecting them to the main thread. The endpoints `c2` and `c2'` of channel C2 (created on Line 5) have session types ($!N.\text{End}$) and ($?N.\text{End}$), respectively. These dual session types express that a number should be sent (!) over `c2` and received (?) over `c2'`. The session types of channel C1 (created on Line 1) are more interesting—they are higher-order. Endpoint `c1` has session type ($!(?N.\text{End}).?N.\text{End}$), which expresses that first a channel of type ($?N.\text{End}$) should be sent, after which a number can be received. The λ-expression of thread T2 captures endpoint `c1`, resulting in the ownership of `c1` being transferred from thread T to thread T2.

The first picture in Figure 2 displays the connectivity graph after both `forks` have been executed, but no other steps have been performed yet. The solid red boxes correspond to threads that are blocked on a receive, while the dotted green boxes correspond to threads that can make progress. The small black arrowheads on the edges indicate the direction of channel ownership: an edge from a thread to a channel indicates that the thread owns an endpoint of that channel, and an edge from

a channel to a channel indicates that an endpoint of the latter channel is stored in one of the buffers of the former channel (an edge between two threads is not possible—all references are *to* channels). The red triangles denote the waiting dependency. A crucial property of the connectivity graph is that the waiting dependency remains acyclic. Acyclicity enables us to find a thread that can make progress by starting at any vertex and repeatedly following the red triangles. For example, when starting at thread T1, which is blocked, we find that thread T2 can perform a step.

When we continue by letting thread T2 perform the `send` operation on Line 6, the `send` will move the endpoint `c2'` into the buffer of C1. In general, the effect of the `send` operation on the connectivity graph is as follows:
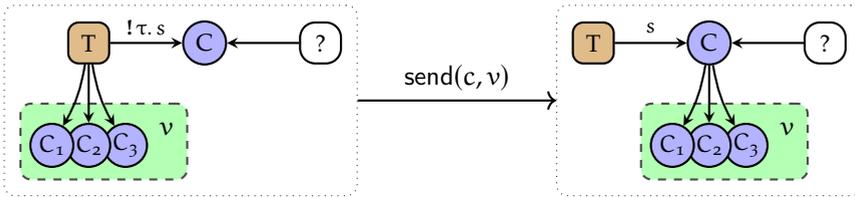
On the left, thread T has ownership of the transmitted value v and the endpoint of carrier channel C with session type $!\tau.\, s$. The value v could in general contain any number of channel references (depicted as C_1, C_2, C_3), for instance when it is a pair of channels or a closure that has captured more than one channel. On the right, we have the resulting connectivity graph that we obtain after the `send` operation has been performed. The session type changes to s (*i.e.,* the remainder of the protocol) and the value v gets transferred to the buffer of C, so the incoming edges of the channels in v are changed from T to C. Note that the red waiting triangles and the information about whether a thread is blocked is not part of the connectivity graph because it can be derived from the run-time configuration. We therefore depict the general transformation rule without waiting triangles and use a neutral color for threads. In our running example, the thread is T2, the channel is C1, and the value v is the single channel C2. The second picture in Figure 2 displays the resulting connectivity graph: the session type of T2 has advanced to ($?\,N.\,\text{End}$) and the incoming edge from C2 to T2 has turned into an incoming edge from C2 to C1.

Next, we let thread T1 perform the `recv` operation on Line 2, which will move the endpoint out of the buffer of channel C1 and bind it to variable `c`. The rule to transform the connectivity graph for a `receive` operation is similar to `send` (the exact rule can be found in Figure 9 in Section 1.3.5). The third picture in Figure 2 displays the resulting connectivity graph, where we see that the session type of T1 advanced, and that the outgoing edge from C1 to C2 has turned into an outgoing edge from T1 to C2. Observe that the connectivity graph has a non-trivial structure—to find a thread that can unblock T2, we need to follow multiple edges to end up in thread T.

PROGRESS AND PRESERVATION   As shown by the examples in Figure 2, connectivity graphs describe the types and abstract reference topology of a program's

execution configuration, but not the concrete expressions and values that constitute the threads and channels. To prove a property about the operational semantics, we need to define a relation that expresses that a configuration ρ is well-formed w.r.t. a connectivity graph G. With that relation at hand, we can carry out a proof in the style of progress and preservation (Wright and Felleisen, 1994; Harper, 2016; Pierce, 2002).

- **Progress**: If ρ is well-formed w.r.t. G, then either ρ is final (all threads have terminated and all channels have been deallocated), or ρ can step (deadlock freedom).

- **Preservation**: If ρ is well-formed w.r.t. G, and can take a step ρ ⤳ ρ′ in the operational semantics, then we can transform G into G′ so that ρ′ is well-formed w.r.t. G′.

It is important to point out that connectivity graphs generalize *heap typings* from traditional progress and preservation proofs for type systems with mutable references (Pierce, 2002; Harper, 2016). Whereas heap typings are flat (they merely give the types of channels, which suffices for type safety), connectivity graphs additionally describe the reference topology and ensure its acyclicity (needed for deadlock and memory leak freedom).[1]

CONNECTIVITY GRAPHS AS A PARAMETRIC PROOF PRINCIPLE   When trying to formalize the above reasoning, we encounter two problems:

1. Due to linear types and concurrency, it is non-trivial to formalize the well-formedness relation of configurations w.r.t. connectivity graphs. Definitions easily end up cluttered with details about *resource separation*, which burdens mechanization in a proof assistant.

2. Proving preservation and progress involves non-trivial reasoning about graphs. For preservation we need to *transform* graphs (to type a post-configuration), and for progress we need to *traverse* graphs (to find a thread that can step). Reasoning about graphs is difficult in a proof assistant because graphs are not inductively defined.

To address these problems, we use separation logic (O'Hearn et al., 2001) as a meta theoretic tool to reason about graphs. Traditionally, separation logic is used as a specification language to write pre- and postconditions for individual programs in Hoare-style logics. Inspired by recent work that uses separation logic to establish type safety using logical relations (Krebbers et al., 2017b; Jung et al., 2018a; Hinrichsen et al., 2021) and intrinsically-typed interpreters and compilers (Rouvoet

---

[1] In fact, in order to ensure that acyclicity is preserved as an invariant, we impose the stronger condition that the *undirected erasure* of the graph is acyclic, *i.e.,*, the graph is an undirected forest when the direction of the edges is erased.

et al., 2020, 2021), we also use separation logic but in the context of a progress and preservation proof.

Our version of separation logic makes it possible to define the well-formedness relation in a way that is local (*i.e.,* talks about threads in isolation) and that hides resources. To adopt separation logic for our connectivity graph, we must decide on what to consider as a resource. The scenarios in Figure 2 suggest that we should consider a vertex's *outgoing edges* as resources, because then a graph transformation, such as the one induced by moving endpoint c2' into C1's buffer, simply amounts to an ownership transfer. To prove preservation, we distill a set of separation logic rules for reasoning about *graph transformations* by simply transferring ownership of resources. To prove progress, we distill a form of *waiting induction* to perform induction on the connectivity graph to find a vertex that can perform a step.

All ingredients of our method (the definition of connectivity graph, the separation logic, the graph transformations, and waiting induction) are parametric in the vertices, edges and labels of the connectivity graph. This is crucial for mechanization: we can encapsulate our proof method as a library that is independent of the concrete programming language. We use our library in combination with the Iris Proof Mode (Krebbers et al., 2017b, 2018)—which provides tactics for separation-logic based reasoning—to effectively hide reasoning about graphs and resources in Coq.

CONTRIBUTIONS   We present a parametric method for proving deadlock and memory leak freedom of binary linear session-typed languages. Concretely:

- We introduce *connectivity graphs* as a generalization of heap typings in progress and preservation proofs. In addition to typing, connectivity graphs track the reference topology.

- We show how to use separation logic in a non-standard way as a language for linking our abstract connectivity graphs to a concrete language's operational semantics and type system.

- We implement connectivity graphs as a library in the Coq proof assistant that is parametric in the type of vertices and edges. Our library includes *graph transformations* as separation logic rules to aid proving preservation, and a principle of *waiting induction* over connectivity graphs to aid proving progress.

- We use our connectivity graph library to obtain the first mechanized proof of deadlock and leak freedom for a binary session-typed λ-calculus with higher-order channels, recursive types, and unrestricted types.

We start by introducing our language (Section 1.2), and explain our key ideas by proving deadlock freedom for it (Section 1.3). Next, we present the parametric aspects of our proof method (Sections 1.4 and 1.5). We then add extensions to our language, and prove a stronger deadlock and memory leak freedom property than the conventional formulation in terms of global progress (Section 1.6), and describe our

Coq mechanization (Section 1.7). We finish with related and future work (Sections 1.8 and 1.9). An archive of the Coq mechanization can be found at Jacobs et al. (2021), and the most recent version at https://github.com/julesjacobs/cgraphs.

1.2 LANGUAGE AND OPERATIONAL SEMANTICS

We present the core of our session-typed λ-calculus with concurrency and asynchronous bidirectional channels (extensions with more features are described in Section 1.6). This language is inspired by GV (Wadler, 2012; Lindley and Morris, 2015), but there are a couple of differences. First, we are more liberal and allow both channel endpoints to be closed anytime, rather than only when a thread terminates. For our proofs this extension poses no problem—it just means that our connectivity graphs might become disconnected. Second, our operational semantics uses a flat thread pool and heap rather than binders and structural congruences, resembling more closely a realistic implementation of message passing. The syntax of expressions of our core language is:

$$e \in \mathsf{Expr} ::= x \mid () \mid n \mid (e, e) \mid \lambda x.\, e \mid c \mid e\, e \mid \mathsf{let}\ () = e\ \mathsf{in}\ e \mid \mathsf{let}\ (x_1, x_2) = e\ \mathsf{in}\ e \mid$$
$$\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{fork}(e) \mid \mathsf{send}(e, e) \mid \mathsf{receive}(e) \mid \mathsf{close}(e) \mid \cdots$$

The literals include the unit value (), numbers $n \in \mathbb{N}$, and channel endpoint references $c \in \mathsf{Chan}$ (these enter expressions at run time, see the operational semantics below). As usual in a linearly-typed language, we consider let-binding constructs let () = e in e and let $(x_1, x_2) = e$ in e for pattern matching on the unit value () and pairs (e, e), respectively.

OPERATIONAL SEMANTICS.   We use an asynchronous semantics with two buffers per channel to guarantee that sends in either direction are non-blocking.[2] This is formally modeled as:

$$c \in \mathsf{Chan} ::= \#(a, t) \qquad\qquad h \in \mathsf{Heap} \triangleq \mathsf{Chan} \xrightarrow{\mathsf{fin}} \mathsf{List}\,\mathsf{Val}$$
$$v \in \mathsf{Val} ::= () \mid n \mid (v, v) \mid \lambda x.\, e \mid c \qquad \rho \in \mathsf{Cfg} \triangleq \mathsf{List}\,\mathsf{Expr} \times \mathsf{Heap}$$

A heap h is a finite map from channel endpoint references to buffers (modeled as lists of values). Channel endpoint references $\#(a, t)$ consist of an address $a \in \mathsf{Addr}$ and a tag $t \in \{0, 1\}$ denoting the endpoint. The operation $\overline{\#(n, t)} \triangleq \#(n, 1 - t)$ gives the opposite endpoint. Configurations $(\vec{e}, h)$ consist of a list of expressions $\vec{e}$, modeling the threads, and a heap h that is shared by these threads. The semantics of most constructs is standard, so we focus on the message passing constructs:

---

2 Due to the session typing discipline, only one of the buffers is expected to be populated at any given time. The two buffers are important to distinguish the origin of the messages, because otherwise an asynchronous send followed by a receive creates a risk that the thread receives back its own message that it just sent.

Pure reduction relation:

$$(\lambda x.\ e)\ v \rightsquigarrow_{\text{pure}} e[v/x]$$
$$\text{let } x = v \text{ in } e \rightsquigarrow_{\text{pure}} e[v/x]$$
$$\text{let } () = () \text{ in } e \rightsquigarrow_{\text{pure}} e$$
$$\text{let } (x_1, x_2) = (v_1, v_2) \text{ in } e \rightsquigarrow_{\text{pure}} e[v_1/x_1][v_2/x_2]$$
$$\text{if } n \text{ then } e_1 \text{ else } e_2 \rightsquigarrow_{\text{pure}} e_1 \qquad\qquad (\text{if } n \neq 0)$$
$$\text{if } n \text{ then } e_1 \text{ else } e_2 \rightsquigarrow_{\text{pure}} e_2 \qquad\qquad (\text{if } n = 0)$$

Head reduction relation:

$$(e_1, h) \rightsquigarrow_{\text{head}} (e_2, h, \epsilon) \qquad\qquad (\text{if } e_1 \rightsquigarrow_{\text{pure}} e_2)$$
$$(\text{fork}(v), h) \rightsquigarrow_{\text{head}} (\#(a, 1), h \uplus \{(a, 0) \mapsto \epsilon, (a, 1) \mapsto \epsilon\}, [v\ \#(a, 0)])$$
$$(\text{if } (a, 0), (a, 1) \notin \text{dom}(h))$$
$$(\text{send}(c, v), h \uplus \{\overline{c} \mapsto \vec{v}\}) \rightsquigarrow_{\text{head}} (c, h \uplus \{\overline{c} \mapsto \vec{v} + [v]\}, \epsilon)$$
$$(\text{receive}(c), h \uplus \{c \mapsto [v] + \vec{v}\}) \rightsquigarrow_{\text{head}} ((c, v), h \uplus \{c \mapsto \vec{v}\}, \epsilon)$$
$$(\text{close}(c), h \uplus \{c \mapsto \epsilon\}) \rightsquigarrow_{\text{head}} ((), h, \epsilon)$$

Global reduction relation:

$$(\vec{e_a} + [K[\ e\ ]] + \vec{e_b}, h) \rightsquigarrow_{\text{global}} (\vec{e_a} + [K[\ e'\ ]] + \vec{e_b} + \vec{e}, h')$$
$$(\text{if } (e, h) \rightsquigarrow_{\text{head}} (e', h', \vec{e}))$$

Evaluation contexts:

$$K \in \text{Ctx} ::= \square \mid (K, e) \mid (v, K) \mid K\ e \mid v\ K \mid$$
$$\text{let } x = K \text{ in } e \mid \text{let } () = K \text{ in } e \mid \text{let } (x_1, x_2) = K \text{ in } e \mid \text{if } K \text{ then } e_1 \text{ else } e_2 \mid$$
$$\text{fork}(K) \mid \text{send}(K, e) \mid \text{send}(v, K) \mid \text{receive}(K) \mid \text{close}(K)$$

Figure 3: The operational semantics of our language.

```
// no counter party
let c1 = fork(λ c1', ())
receive(c1)
```



```
// protocol violation
let c1 = fork(λ c1', receive(c1'); ...)
receive(c1)
```



```
// circular dependency
let c1 = fork(λ c1', send(c1', c1'))
let (c1,c1') = receive(c1)
let c2 = fork(λ c2',
    receive(c2'); send(c1', 2); ...)
receive(c1); send(c2, 3)
```



```
// memory leak
let c1 = fork(λ c1', ())
let c2 = fork(λ c2', ())
send(c2, c1)
send(c1, c2)
```
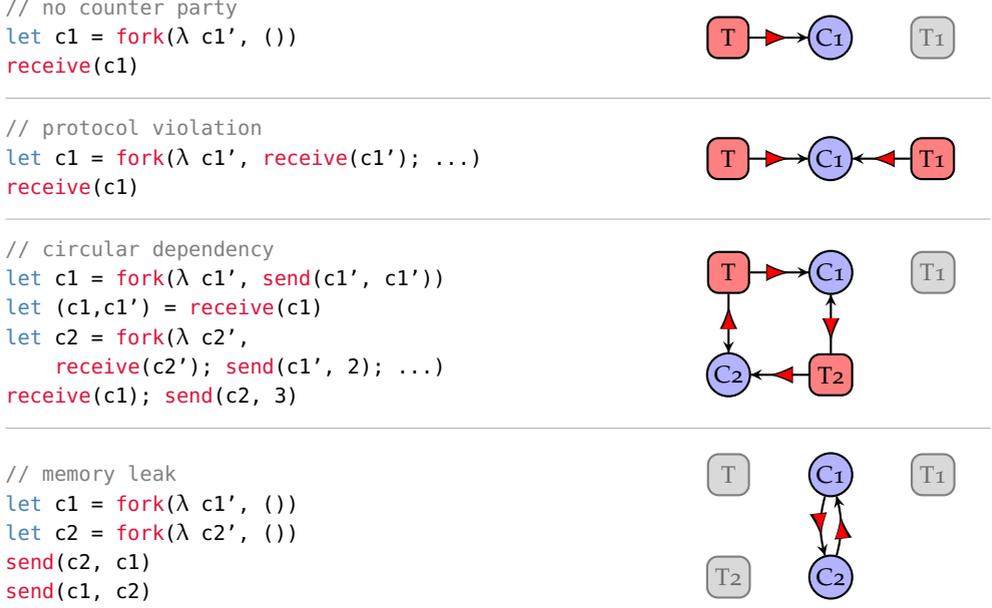

Figure 4: Examples of configurations that are deadlocked or have leaked.

$\mathsf{fork}(v)$ Allocates a new channel with endpoints $c_{\mathsf{left}} \triangleq \#(a, 0)$ and $c_{\mathsf{right}} \triangleq \#(a, 1)$, where a is a fresh address. It starts a new thread running $v\, c_{\mathsf{left}}$ (v should be a function) and returns c_{right}.

$\mathsf{send}(c, v)$ Places the message v into the buffer of the opposite endpoint $\overline{c}$ of c and returns c.[3] This construct does not block.

$\mathsf{receive}(c)$ Takes a message v out of the buffer of endpoint c and returns the pair (c, v). If the buffer is empty, it blocks until a message is available.

$\mathsf{close}(c)$ Closes the endpoint c and returns the unit value $()$. This construct does not block.

The formal definition of the semantics is given in Figure 3. It involves three reduction relations: (1) pure reductions $e \rightsquigarrow_{\mathsf{pure}} e'$, (2) head-reductions $(e, h) \rightsquigarrow_{\mathsf{head}} (e', h', \vec{e})$, where $\vec{e}$ is a list of spawned threads (empty for all constructs but fork), and (3) global reductions $\rho \rightsquigarrow_{\mathsf{global}} \rho'$. Global reductions make use of standard call-by-value evaluation contexts $K \in \mathsf{Ctx}$.

DEADLOCKS AND MEMORY LEAKS. Untyped programs in our language can deadlock or have memory leaks. A configuration $(\vec{e}, h)$ is *deadlocked* if each expression

---

3 The reason why send returns the endpoint c is the session type system, which gives the endpoint a new type, prescribing the remainder of the protocol. The same applies to the receive operation.

$$\frac{\Gamma = \{x \mapsto \tau\}}{\Gamma \vdash x : \tau} \qquad \frac{.}{\emptyset \vdash () : \mathbf{1}} \qquad \frac{n \in \mathbb{N}}{\emptyset \vdash n : \mathbf{N}} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{1} \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } () = e_1 \text{ in } e_2 : \tau}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2 \uplus \{x_1 \mapsto \tau_1\} \uplus \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } (x_1, x_2) = e_1 \text{ in } e_2 : \tau}$$

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \qquad \Gamma_2 \vdash e_2 : \tau \qquad \Gamma_2 \vdash e_3 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \qquad \frac{\Gamma \vdash e : \overline{s} \multimap \mathbf{1}}{\Gamma \vdash \text{fork}(e) : s}$$

$$\frac{\Gamma_1 \vdash e_1 : !\tau.\, s \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{send}(e_1, e_2) : s} \qquad \frac{\Gamma \vdash e : ?\tau.\, s}{\Gamma \vdash \text{receive}(e) : s \times \tau} \qquad \frac{\Gamma \vdash e : \text{End}}{\Gamma \vdash \text{close}(e) : \mathbf{1}}$$

Figure 5: The static type system of our language.

$e \in \vec{e}$ is a receive(c) that is waiting on an empty buffer c in the heap h. A configuration $(\vec{e}, h)$ has *leaked* if each expression $e \in \vec{e}$ is a value, but the heap h is not empty, meaning not all channels have been closed. In Figure 4 we show examples of both. On the left we show the code, and on the right we show a graphical representation of the resulting configuration. As in Section 1.1, boxes denote threads (*i.e.,* expressions), circles denote channels (*i.e.,* buffer pairs), black arrows denote channel references, and red triangles denote the waiting dependency. Concretely, a thread with a red triangle pointing to a channel is waiting to receive a message from that channel, and a channel with a red triangle pointing to a thread is waiting for the thread to send a message along that channel.

The simplest form of deadlock is a thread attempting to receive a message from a channel that nobody else has a reference to (first program). If threads violate the usual protocol that one side receives and the other side sends a message, then a deadlock can occur if both try to receive (second program). A deadlock can occur even if all parties are locally well behaved, but cause a cyclic dependency (third program). Note that even though the reference structure (black arrows) of this example forms a directed acyclic graph, a deadlock occurs because the waiting direction (red triangles) can be opposite of the reference direction (black arrows). Finally, memory leaks can occur if channels are not properly closed (fourth program).

SESSION TYPING.   A linear type system with session types can be used to rule out deadlocks:[4]

$$\tau \in \text{Type} ::= \mathbf{1} \mid \mathbf{N} \mid \tau \times \tau \mid \tau \multimap \tau \mid s$$
$$s \in \text{Session} ::= \text{End} \mid ?\tau.\,s \mid !\tau.\,s$$

A session type s denotes a sequence of actions, with $?\tau$ indicating a receive, $!\tau$ a send, and End termination, where τ denotes the type of the message. The *dual* $\overline{s}$ of a session type s is defined by flipping all sends (!) and receives (?):

$$\overline{\text{End}} \triangleq \text{End} \qquad\qquad \overline{!\tau.\,s} \triangleq ?\tau.\,\overline{s} \qquad\qquad \overline{?\tau.\,s} \triangleq !\tau.\,\overline{s}$$

The rules of the type system are shown in Figure 5. Note that the type system is *higher-order* because it allows sending any value over a channel, including functions and channel endpoints.

Session types ensure deadlock and leak freedom by combining channel and thread creation through the fork construct.[5] Together with linear channel typing, this ensures that the resulting reference structure is *acyclic*, even when viewed as an *undirected graph*, in which edges may be traversed in either direction. Let us consider the deadlocked programs in Figure 4. The first one is ruled out by ensuring that there always exists a counter party (due to the absence of weakening). The second one is ruled out by ensuring that all threads adhere to protocols (due to session duality). The third one is ruled out by ensuring that the reference structure is acyclic (due to the absence of contraction). The memory leak in the last example is ruled out by a combination of these rules.

1.3 KEY IDEAS

Before we detail the abstractions that make our proof method parametric (Section 1.4 and Section 1.5), we describe a concrete instantiation of our method to our session-typed language (Section 1.2). To do so, we first discuss the well-known method of progress and preservation and the challenges in applying it to prove deadlock and resource leak freedom (Section 1.3.1). To address these challenges, we introduce connectivity graphs (Section 1.3.2) and describe how we use separation logic to formalize run-time typing judgments for individual expressions (Section 1.3.3) and a well-formedness predicate for configurations (Section 1.3.4). We finally show how to use our proof method to prove preservation (Section 1.3.5) and progress (Section 1.3.6).

---

4 For simplicity we let even integers be linear; in Section 1.6 we extend the type system with support for unrestricted types.

5 If we had a new_chan : $\mathbf{1} \multimap (s \times \overline{s})$ construct, then let (c_1, c_2) = new_chan () in let x = receive(c_1) in ... would deadlock.

1.3.1 *Generalizing The Progress and Preservation Method*

Traditionally, a language is said to be *type safe* if well typed programs do not get stuck. For purely functional languages, like the Simply Typed Lambda Calculus (STLC), this is stated as:

Theorem 1.3.1 (Type safety). *If $\emptyset \vdash e : \tau$ and $e \leadsto^* e'$, then either e' is a value, or e' can step further (i.e., $\exists e''. e' \leadsto e''$).*

Type safety is often proved using the method of *progress and preservation* (Wright and Felleisen, 1994; Harper, 2016; Pierce, 2002), which decomposes type safety into two properties that imply it:

- **Preservation:** If $\emptyset \vdash e : \tau$ and $e \leadsto e'$, then $\emptyset \vdash e' : \tau$.

- **Progress:** If $\emptyset \vdash e : \tau$, then either e is a value, or e can step (i.e., $\exists e''. e' \leadsto e''$).

For pure languages like STLC, the proofs of these properties are straightforward: both properties are proved by induction on the structure of the typing judgment and/or the reduction relation.

For languages with mutable state or concurrency, the above properties must be generalized to account for a program's run-time configurations. In general, for a language with expressions Expr we have a set $\rho \in$ Cfg of configurations, an initial configuration init $\in$ Expr $\rightarrow$ Cfg, and a predicate final $\in$ Cfg $\rightarrow$ Prop of configurations that are considered to be safely terminated.

Theorem 1.3.2 (Generalized type safety). *If $\emptyset \vdash e : ()$ and* init$(e) \leadsto^* \rho$*, then either* final(ρ) *or ρ can step further (i.e., $\exists \rho'. \rho \leadsto \rho'$).*

Recall that for our session-typed language we have Cfg $\triangleq$ List Expr $\times$ Heap. We let final select configurations where all threads have terminated with a unit value, and the heap is empty:

$$\text{init}(e) \triangleq ([e], \emptyset) \qquad\qquad \text{final}(\vec{e}, h) \triangleq h = \emptyset \land \forall i.\, e_i = ()$$

By defining final this way, the type safety theorem expresses *deadlock and memory leak freedom*.[6] To see why, consider a configuration $\rho = (\vec{e}, h)$ that does not satisfy final(ρ) and cannot step any further. It must either consist of threads $\vec{e}$ that have not terminated but cannot step (indicating a deadlock), or of terminated threads $\vec{e}$ but a non-empty heap h (indicating a memory leak).

For deadlock and resource leak freedom, we need to restrict expressions to have a ground type. For example, an expression like fork$(\lambda x.\, \text{close}(x)) : \text{End}$ exhibits a trivial memory leak because the channel endpoint returned by fork() is still active.

---

6 This kind of deadlock freedom is also known as *global progress* in the session type literature. In we prove a stronger property that also rules out partial deadlocks.

For simplicity, we use the unit type () in Theorem 1.3.2, but of course, other ground types like **N** would suffice too.

The method of progress and preservation can be generalized to prove our generalized type safety theorem (Theorem 1.3.2) by choosing a well-formedness predicate wf $\in$ Cfg $\to$ Prop that satisfies:

- **Initialization:** If $\emptyset \vdash e : ()$, then wf(init(e)).

- **Generalized preservation:** If wf(ρ) and $\rho \rightsquigarrow \rho'$, then wf($\rho'$).

- **Generalized progress:** If wf(ρ), then either final(ρ) or ρ can step further (*i.e.,* $\exists \rho'. \rho \rightsquigarrow \rho'$).

The primary challenge is to define a well-formedness predicate wf in such a way that these properties can be proved. A naive definition of wf($\vec{e}, h$) would simply demand each expression in the thread pool $\vec{e}$ and each buffer in the heap h to be well typed. Unfortunately, this naive definition does not quite work:

1. Channel references $\#(a, t)$ enter the expressions $\vec{e}$ and heap h throughout the execution of the program. The typing judgment $\Gamma \vdash e : \tau$ of our type system (Figure 5) does not account for channel references $\#(a, t)$ because they cannot be written in source programs.

2. Whenever a channel reference $\#(a, t)$ appears in a thread or channel buffer, the type of $\#(a, t)$ should match up with the values in the buffers at address a in the heap h, and with the type $\#(a, 1 - t)$ of the other endpoint.

For the simpler case of proving type safety for the STLC with references, Harper (2016) and Pierce (2002) remedy issue (1) by introducing a *run-time typing judgment* $\Gamma; \Sigma \vdash e : \tau$. This judgment extends the static typing judgment with a *heap typing* Σ, which assigns types to heap addresses. Issue (2) is addressed because the heap typing makes sure that the typing of each reference is consistent with the corresponding value in the heap.

Unfortunately, adapting this approach to prove deadlock and resource freedom is not as simple. Conventional heap typings only capture the typing of addresses, not the acyclicity of the reference topology. The latter is crucial to prove "generalized progress", which states that the well-formedness predicate wf indeed implies deadlock and resource leak freedom.

1.3.2 *Generalizing Heap Typings to Connectivity Graphs*

Our notion of *connectivity graphs* generalizes the notion of heap typings by simultaneously keeping track of the types of channels in the heap, and providing an abstract representation of the reference topology. In their full generality, connectivity
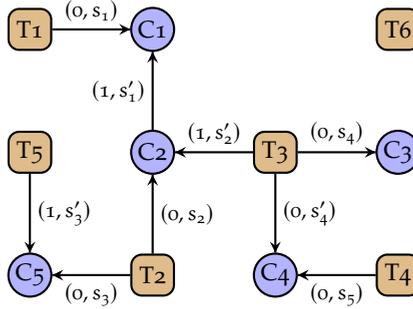
Figure 6: An example of a connectivity graph. Brown boxes depict threads, and blue circles depict channels.

graphs are represented as finite maps from pairs of vertices V to the labels L on the edges between them:

$$G \in \mathsf{Cgraph}(V, L) \triangleq \left\{ G \in V \times V \xrightarrow{\mathsf{fin}} L \,\middle|\, G \text{ has no undirected cycles} \right\}$$

To reason about our language defined in Section 1.2, we instantiate the vertices V and edge labels L of a connectivity graph $\mathsf{Cgraph}(V, L)$ as follows:

$$v \in V ::= \mathsf{Thread}(i) \mid \mathsf{Chan}(a) \qquad l \in L \triangleq \{0, 1\} \times \mathsf{Session}$$

The vertices V are threads $\mathsf{Thread}(i)$ (with position i in the thread pool) or channels $\mathsf{Chan}(a)$ (with address a in the heap). The edges are channel references and have a label $(t, s) \in L$ that consists of a tag $t \in \{0, 1\}$, indicating the channel endpoint pointed to, and a session type $s \in \mathsf{Session}$, indicating the endpoint's session type. An example of a connectivity graph is depicted in Figure 6. Note that the red triangles that we used to depict the waiting directions in Figures 2 and 4 in Section 1.1 are not part of the connectivity graph itself, because these can be derived from the run-time configuration. The session types on the edges *are* part of the connectivity graph, because they cannot be derived from the run-time configuration. In Theorem 1.3.4 we formalize how the red triangles are derived.

Connectivity graphs corresponding to configurations in our language have a number of important properties. First, vertices $\mathsf{Thread}(i)$ can have an arbitrary number of outgoing edges, but have no incoming edges. That is because threads can own channel endpoints, but threads can never be owned. Second, vertices $\mathsf{Chan}(a)$ can also have an arbitrary number of outgoing edges, but at most two incoming edges. Outgoing edges are due to higher-order channels—a channel c_1 can be sent over another channel c_2, resulting in an edge from c_2 to c_1 that models that c_2 owns c_1. Ingoing edges correspond to a channel's endpoints, which can be at most two. The number of incoming edges is one in case one channel endpoint has been deallocated, but the other is still in use.

A key ingredient of connectivity graphs is the acyclicity restriction. They should be *acyclic* in the *undirected* sense: there must be no cycles even if we disregard the direction of the edges. In other words, a connectivity graph must be an unrooted undirected forest if we erase the direction of the edges. The third example in Figure 4 shows why acyclicity in the undirected sense is important.

To formally reason about ownership, we introduce the following functions:

$$\text{out}(G, \nu) \in V \xrightarrow{\text{fin}} L \qquad\qquad \text{in}(G, \nu) \in \text{Multiset } L$$

The outgoing edges $\text{out}(G, \nu)$ determine which *resources vertex ν owns*, whereas the incoming edges $\text{in}(G, \nu)$ determine at which labels (*i.e.,* types) the *vertex ν is owned*. We use the above functions in the definitions of the run-time typing judgment (Section 1.3.3) and the configuration well-formedness predicate (Section 1.3.4). We represent the the outgoing edges $\text{out}(G, \nu)$ of a vertex ν as a finite map from vertices to labels to track *which* resources a vertex ν owns and at *which type*. The incoming edges $\text{in}(G, \nu)$ of a vertex ν, however, we represent as a multiset of labels because it only matters at *which type* a vertex ν is owned, but not by whom (note that only channel endpoints can be owned).

WHAT IS PARAMETRIC IN THIS SECTION   Connectivity graphs $\text{Cgraph}(V, L)$ are parametric over the type of vertices V and labels L. All theory about connectivity graphs (including the separation logic) that we present throughout the rest of this section is parametric. Connectivity graphs and their theory are thus modularly separated from the operational semantics and type system of the language, which we found to be essential for keeping the complexity of the mechanization manageable.

1.3.3 *Run-Time Typing Judgment Using Separation Logic*

In the previous section, we developed the notion of a connectivity graph as a generalization of the heap typing, known from type safety proofs of the STLC with references (Harper, 2016; Pierce, 2002). We now make this generalization precise, develop a run-time typing judgment for our language, and show how we can use separation logic to hide reasoning about linearity.

We start with the run-time judgment $\Gamma; \Sigma \vdash e : \tau$, where $\Sigma \in V \xrightarrow{\text{fin}} L$ provides the session types of the free channel references in e. Channel references amount to edges in our connectivity graph, and thus Σ becomes the set of *outgoing edges* $\text{out}(G, \nu)$ associated with the threads and channels ν occurring in e. Let us consider the typing rule for channel references and function application:

$$\frac{\cdot}{\emptyset; \{\text{Chan}(a) \mapsto (t, s)\} \vdash \#(a, t) : s} \qquad \frac{\Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2; \Sigma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2; \Sigma_1 \uplus \Sigma_2 \vdash e_1 \; e_2 : \tau_2}$$

$$(\text{Emp})(\Sigma) \triangleq (\Sigma = \emptyset)$$

$$(P, Q \in \text{iProp} \triangleq (V \xrightarrow{\text{fin}} L) \rightarrow \text{Prop})$$

$$(\ulcorner \phi \urcorner)(\Sigma) \triangleq \phi \wedge (\Sigma = \emptyset)$$

$$(\Sigma \in V \xrightarrow{\text{fin}} L)$$

$$(P \wedge Q)(\Sigma) \triangleq P(\Sigma) \wedge Q(\Sigma)$$

$$(\exists x. P(x))(\Sigma) \triangleq \exists x. P(x)(\Sigma)$$

$$(\text{own}(\Sigma'))(\Sigma) \triangleq (\Sigma = \Sigma')$$

$$(P * Q)(\Sigma) \triangleq \exists \Sigma_1 \Sigma_2. \, \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) = \emptyset \wedge \Sigma = \Sigma_1 \uplus \Sigma_2 \wedge P(\Sigma_1) \wedge Q(\Sigma_2)$$

$$(P \twoheadrightarrow Q)(\Sigma) \triangleq \forall \Sigma'. \, \big(\text{dom}(\Sigma) \cap \text{dom}(\Sigma') = \emptyset \wedge P(\Sigma')\big) \Rightarrow Q(\Sigma \uplus \Sigma')$$

Figure 7: Semantics of separation logic.

Because our language is linear, we insist that the Σ-context is a singleton in the rule for channel references. In the rule for application, both contexts are split into disjoint parts for the subexpressions. Unfortunately, this leads to a multiplication of contexts and disjointness side conditions, because of the disjoint unions in the conclusion. These side conditions cannot be ignored because we want to mechanize our results. This is not a big issue for the variable context Γ since we mostly deal with *closed* expressions (*i.e.,* with $\Gamma = \emptyset$) because the operational semantics operates on closed expressions. The Σ-context, however, is in general non-empty for run-time expressions.

We use separation logic (O'Hearn et al., 2001) to hide the Σ-context and its disjointness conditions. We work with separation logic propositions iProp $\triangleq (V \xrightarrow{\text{fin}}$ L) $\rightarrow$ Prop, which are predicates over a context $\Sigma \in V \xrightarrow{\text{fin}} L$ of outgoing edges. Our use of separation logic as an *internal*, meta theoretical tool is inspired by Rouvoet et al. (2020) and contrasts with traditional uses which employ separation logic as an *external*, user-visible tool when specifying programs in Hoare-style logics. The separation logic connectives are defined in Figure 7. To assert that a separation logic proposition P is true, is to assert that $P(\emptyset)$ is true. An important special case is that $P \twoheadrightarrow Q$ is true, if $\forall \Sigma. \, P(\Sigma) \Rightarrow Q(\Sigma)$.

Instead of the ordinary typing judgment $(\Gamma; \Sigma \vdash e : \tau) \in \text{Prop}$ we define a separation-logic based judgment $(\Gamma \vdash e : \tau) \in \text{iProp}$, so that $(\Gamma; \Sigma \vdash e : \tau)$ iff $(\Gamma \vdash e : \tau)(\Sigma)$. The preceding two rules are then reformulated as follows:

$$\frac{\text{own}(\text{Chan}(a) \mapsto (t, s))}{\emptyset \vdash \#(a, t) : s} *$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \quad * \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1 \, e_2 : \tau_2} *$$

The Σ-contexts are hidden by the separation logic connectives, and the disjointness conditions on Σ are taken care of by the separating conjunction ($*$). At a channel reference, we use the $\text{own}(\Sigma)$ connective, which asserts that the separation logic resource is precisely Σ.

$$\frac{\ulcorner \Gamma = \{x \mapsto \tau\}\urcorner}{\Gamma \vdash x : \tau}* \qquad \frac{\text{Emp}}{\emptyset \vdash () : \mathbf{1}}* \qquad \frac{\ulcorner n \in \mathbb{N}\urcorner}{\emptyset \vdash n : \mathbf{N}}* \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \quad * \quad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2}*$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \quad * \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\ e_2 : \tau_2}* \qquad \frac{\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}*$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad * \quad \Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}* \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{1} \quad * \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } () = e_1 \text{ in } e_2 : \tau}*$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \quad * \quad \Gamma_2 \uplus \{x_1 \mapsto \tau_1\} \uplus \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{let } (x_1, x_2) = e_1 \text{ in } e_2 : \tau}*$$

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \quad * \quad (\Gamma_2 \vdash e_2 : \tau \quad \wedge \quad \Gamma_2 \vdash e_3 : \tau)}{\Gamma_1 \uplus \Gamma_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}* \qquad \frac{\Gamma_1 \vdash e_1 : !\tau.s \quad * \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{send}(e_1, e_2) : s}*$$

$$\frac{\Gamma \vdash e : ?\tau.s}{\Gamma \vdash \text{receive}(e) : s \times \tau}* \quad \frac{\Gamma \vdash e : \bar{s} \multimap \mathbf{1}}{\Gamma \vdash \text{fork}(e) : s}* \quad \frac{\Gamma \vdash e : \text{End}}{\Gamma \vdash \text{close}(e) : \mathbf{1}}* \quad \frac{\text{own}(\text{Chan}(a) \mapsto (t, s))}{\emptyset \vdash \#(a, t) : s}*$$

Figure 8: The separation-logic based run-time type system of our language.

An exception to the rule that contexts are split up disjointly (with $*$) is the if e_1 then e_2 else e_3 expression. Although the channel references occurring in e_1 must be disjoint from those occurring in e_2 and e_3, the same endpoint is allowed to occur in both e_2 and e_3, because only one of the branches will be executed. This pattern too fits neatly in the separation logic methodology; we use separating conjunction ($*$) between e_1 and e_2, e_3, but ordinary conjunction ($\wedge$) between e_2 and e_3:

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \quad * \quad (\Gamma_2 \vdash e_2 : \tau \quad \wedge \quad \Gamma_2 \vdash e_3 : \tau)}{\Gamma_1 \uplus \Gamma_2 \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}*$$

Figure 8 contains the full definition of our run-time type system using separation logic. Although it is possible to define the meaning of general inductive separation logic inference rules via Tarski's fixed point theorem, this generality is not necessary here: the expressions in the premises of each rule are strictly smaller than the expression in the conclusion, so the rules can be interpreted as being defined by recursion on the expression. We use this approach in the Coq formalization, because it has the additional benefit that we get the inversion rules for free.

A key strength of separation logic is that we can prove assertions using the proof rules of the logic of Bunched Implications (BI) (O'Hearn and Pym, 1999).

For example, separating conjunction ($*$) is associative and commutative, separating conjunction ($*$) has Emp as identity element, and magic wand ($-*$) is the adjoint of separating conjunction ($*$). We use the Iris Proof Mode (Krebbers et al., 2017b, 2018) to reason abstractly using the rules of separation logic in Coq (see Section 1.7 for details).

WHAT IS PARAMETRIC IN THIS SECTION. The definition of the separation logic connectives and the proof rules for the separation logic are parametric in the types of vertices V and labels L.

1.3.4 *Well-Formedness of Configurations Using Connectivity Graphs*

Now that we have a run-time typing judgment for a single expression, we are in a position to define which configurations are well-formed. Recall that a configuration is a pair $(\vec{e}, h)$ where $\vec{e}$: List Expr is the thread pool and where h : Chan $\xrightarrow{\text{fin}}$ List Val is the heap of channel buffers. We must certainly insist that all threads $\vec{e}$ are well-typed expressions (of unit type), and that all the values inside the heap h are well-typed. For the latter we have to ensure that a channel's endpoints are of dual types, modulo the messages queued up in the buffer. This requires us to consider the *incoming edges* $\text{in}(G, v)$ of a vertex v in addition to its *outgoing edges* $\text{out}(G, v)$. We can thus state well-formedness of a configuration in terms of its connectivity graph.

A configuration is well-formed if there exists a connectivity graph such that each thread and channel is locally well-formed with respect to its vertex in the graph:

$$\text{wf}(\vec{e}, h) \triangleq \exists G : \text{Cgraph}(V, L). \forall v \in V. \text{wf}^{local}_{(\vec{e}, h)}(v, \text{in}(G, v))(\text{out}(G, v))$$

Here, $\text{wf}^{local}_{(\vec{e}, h)}$: $V \times$ Multiset $L \to$ iProp gives the *local well-formedness condition* for each vertex. It has two explicit arguments (the vertex $v \in V$ and its incoming edges $\text{in}(G, v) \in$ Multiset L), and an extra argument $\text{out}(G, v) \in V \xrightarrow{\text{fin}} L$ that will form the vertex' local Σ-context, which is implicit in the type signature of wf^{local} because iProp $\triangleq (V \xrightarrow{\text{fin}} L) \to$ Prop.

A thread is locally well-formed if it is well-typed (with the implicit Σ-context given by its outgoing edges), and has no incoming edges (because threads cannot be owned):

$$\text{wf}^{local}_{(\vec{e}, h)}(\text{Thread}(i), \Delta) \triangleq \begin{cases} \ulcorner \Delta = \emptyset \urcorner * \emptyset \vdash e_i : \mathbf{1} & \text{if } i < |\vec{e}| \\ \ulcorner \Delta = \emptyset \urcorner & \text{otherwise} \end{cases}$$

Note that wf quantifies over any vertex $v \in V$, and we thus have to consider any thread index i, including those $i \geqslant |\vec{e}|$ that are not yet in use. For such indexes, we use the separation logic proposition $\ulcorner \Delta = \emptyset \urcorner$ to assert that both the incoming and

outgoing edges are empty. The latter is implicit by the semantics of $\ulcorner \Delta = \emptyset \urcorner$ (see [Figure 7](#)).

A channel is locally well-formed if the buffers are well-typed (with the implicit Σ-context given by its outgoing edges), and have matching incoming edges match the types of the endpoints:

$$\mathsf{wf}^{local}_{(\vec{e},h)}(\mathsf{Chan}(a), \Delta) \triangleq \begin{cases} \exists s_0, s_1, s. \ulcorner \Delta = \{(0, s_0), (1, s_1)\} \urcorner * & \text{if } \#(a, 0), \#(a, 1) \in \mathrm{dom}(h) \\ \quad \vdash_{buf} h(a, 0) : (s_0, s) * & \\ \quad \vdash_{buf} h(a, 1) : (s_1, \overline{s}) & \\ \exists b, s. \ulcorner \Delta = \{(t, s)\} \urcorner * & \text{if } \#(a, t) \in \mathrm{dom}(h) \\ \quad \vdash_{buf} h(a, t) : (s, \mathsf{End}) & \text{and } \#(a, 1 - t) \notin \mathrm{dom}(h) \\ \ulcorner \Delta = \emptyset \urcorner & \text{otherwise} \end{cases}$$

In this definition we have to consider three cases. The first case corresponds to the situation in which both buffers are still in use. In that case, there must be two incoming edges in the connectivity graph, labeled with session types that are dual modulo the values in the buffers. For instance, if the left endpoint has session type $?\tau_1.?\tau_2.s$ and the right endpoint has session type $\overline{s}$, then the buffer of the left endpoint must be $[v_1, v_2]$ with $\vdash v_1 : \tau_1$ and $\vdash v_2 : \tau_2$. The second case corresponds to the situation in which one buffer has been deallocated. The third case corresponds to the situation in which the channel is not allocated (or both buffers have been deallocated).

The *buffer typing judgment* $\vdash_{buf} \vec{v} : (s_1, s_2)$ is inductively defined by the following rules:

$$\frac{}{\vdash_{buf} \epsilon : (s, s)}{}^{\mathsf{Emp}}* \qquad \frac{\emptyset \vdash v : \tau \quad * \quad \vdash_{buf} \vec{v} : (s_1, s_2)}{\vdash_{buf} ([v] \mathbin{+\!\!+} \vec{v}) : (?\tau.s_1, s_2)}*$$

These rules express that $\vdash_{buf} \vec{v} : (s_1, s_2)$ holds if s_1 is equal to prefixing s_2 with the types of the values $\vec{v}$ in the buffer. Note that similar to the other run-time judgments, the buffer typing judgment is defined in separation logic, which implicitly ensures that the Σ-environment is distributed disjointly over the values in the buffer.

WHAT IS PARAMETRIC IN THIS SECTION. The definition of wf is parametric in the type of vertices V and labels L, but also a *local well-formedness predicate* wf^{local} that captures the language-specific information by linking the incoming and outgoing edges of each vertex to their run-time counterpart.

1.3.5 *Proving Preservation Using Local Graph Transformations*

Now that we have defined the well-formedness predicate $\mathsf{wf}(\vec{e}, h)$, we must prove that it is preserved by the operational semantics: if $(\vec{e}, h) \leadsto_{\text{global}} (\vec{e}', h')$, then $\mathsf{wf}(\vec{e}, h)$
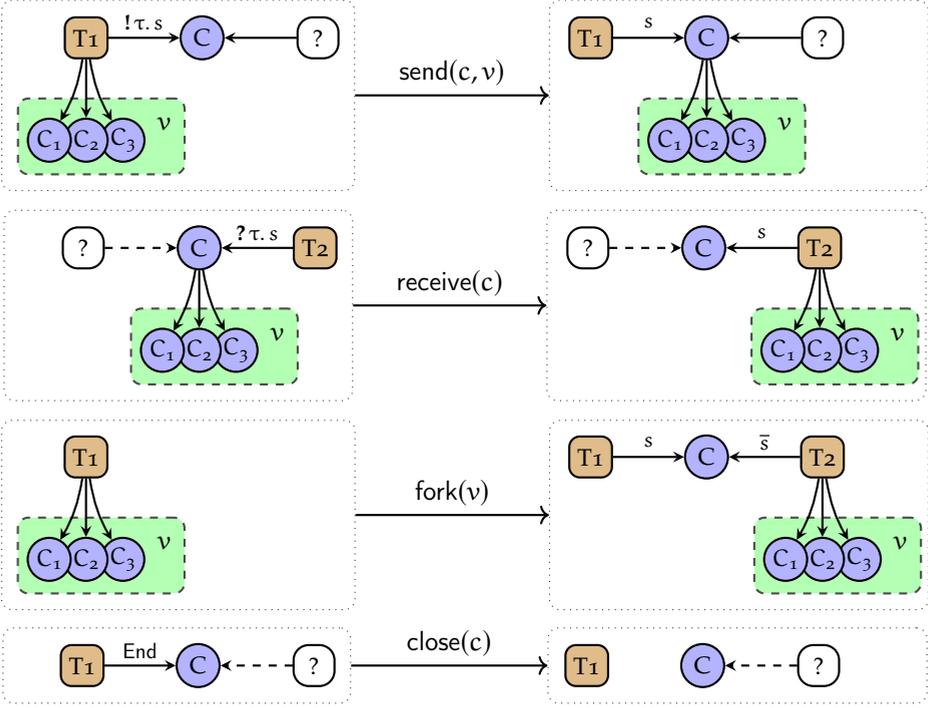
Figure 9: The operational steps and the corresponding connectivity graph transformations.

implies $\mathsf{wf}(\vec{e}', h')$. Recall that the well-formedness predicate $\mathsf{wf}(\vec{e}, h)$ intuitively means "there exists a connectivity graph G describing the configuration $(\vec{e}, h)$", so when the configuration steps to a new configuration $(\vec{e}', h')$, we must show that there exists a new connectivity graph G' that describes $(\vec{e}', h')$.

If the head step is a pure step, then the new connectivity graph is exactly the same as the old one, and the preservation of the well-formedness predicate follows by a standard case analysis of the possible pure steps, because the heap does not change and no new threads are spawned.

Operational steps that involve channel operations are the interesting cases because they may alter the connectivity graph. Figure 9 provides a schematic overview. We focus on the $\mathsf{send}(c, v)$ operation, which moves the value v from the thread into the buffer of channel c. The session type in the label on the edge corresponding to c itself must change from $!\tau. s$ to s. Additionally, if the value v contains channel references, the connectivity graph must change to reflect this. The changes to the connectivity graph for send and the other channel operations are depicted in Figure 9.

Once we have chosen the appropriate new connectivity graph G', we have to prove that this graph indeed describes the new configuration $(\vec{e}', h')$. This amounts to showing that the local well-formedness condition $\mathsf{wf}^{local}_{(\vec{e}',h')}(v, \mathsf{in}(G', v))(\mathsf{out}(G', v))$ is re-established for every vertex v.

For send(c, v) the parts of the $(\vec{e}, h)$-configuration and $(\vec{e}', h')$-configuration are:

$$e_i = K[\,\mathsf{send}(c, v)\,] \qquad\qquad h(\overline{c}) = \vec{v}$$
$$e'_i = K[\,c\,] \qquad\qquad h'(\overline{c}) = \vec{v} +\!\!+ [v]$$

The thread pool and heap do not change at other locations. After this change to the configuration and the corresponding change to the connectivity graph (as depicted in Figure 9), we classify the vertices into three types and explain how the local well-formedness $\mathsf{wf}^{local}_{(\vec{e}', h')}$ is restored.

1. For the vertices v' where neither the corresponding part of the configuration nor the incoming and outgoing edges change, $\mathsf{wf}^{local}_{(\vec{e}, h)}(v', \mathsf{in}(G, v'))(\mathsf{out}(G, v'))$ remains valid.

2. For the vertices v' that correspond to channels referenced *inside the message v*, the owner changes from Thread(i) to Chan$(c.1)$ (corresponding to T1 and C in the figure). These vertices are not affected either, because $\mathsf{in}(G, v') = \mathsf{in}(G', v')$. Since $\mathsf{in}(G, v')$ and $\mathsf{in}(G', v')$ are multisets of *labels*, they are thus not affected by the change of owner.

3. The vertices $v_1 = \mathsf{Thread}(i)$ and $v_2 = \mathsf{Chan}(c.1)$ (corresponding to T1 and C in the figure) are the vertices that are truly affected. Re-establishing their $\mathsf{wf}^{local}_{(\vec{e}', h')}(v_{12}, \mathsf{in}(G', v_{12}))(\mathsf{out}(G', v_{12}))$ requires some language-specific reasoning, because both their part of the configuration and their incoming and outgoing edges change.

There is another proof obligation that we need to meet: the connectivity graph has to remain acyclic when we do these local transformations.

Even though Figure 9 looks hopelessly language specific, we show that we can use our separation logic to distill abstract rules for *local graph transformations* (Section 1.5). These rules involve the transfer of resources between the old local well-formedness predicates $\mathsf{wf}^{local}_{(\vec{e}, h)}(v, \mathsf{in}(G, v))$ and the new local well-formedness predicates $\mathsf{wf}^{local}_{(\vec{e}', h')}(v, \mathsf{in}(G', v))$ (for the affected vertices v in question). To distill these rules, it is crucial that the local well-formedness predicate is a separation logic proposition, which enables reasoning using the abstract proof rules of separation logic, without explicitly having to reference the graph, nor having to explicitly establish acyclicity, nor having to deal with disjointness conditions. The reasoning left to the user of the rule is purely local and precisely the language-specific reasoning that cannot be done generically. The result is that the preservation proof appears to perform no graph reasoning at all: at no point in the preservation proof is there any value of type $G, G' : \mathsf{Cgraph}(V, L)$ in the proof context.

WHAT IS PARAMETRIC IN THIS SECTION.   The separation-logic based rules for *local graph transformations* (Section 1.5) are parametric in the type of vertices V and labels L, and the local well-formedness predicate.

1.3.6 *Proving Progress Using Waiting Induction*

To prove progress, we have to show that if $\mathsf{wf}(\vec{e}, h)$ holds, then either $\mathsf{final}(\vec{e}, h)$ holds (*i.e.*, $e_i = ()$ for all i and $h = \emptyset$), or the configuration can step. This is equivalent to saying that:

$$\mathsf{wf}(\vec{e}, h) \text{ and } \mathsf{active}(\vec{e}, h) \neq \emptyset \text{ implies that } (\vec{e}, h) \text{ can step}$$

Here, $\mathsf{active}(\vec{e}, h)$ is the set of threads and channels that have not yet terminated and not yet been fully deallocated, respectively.

Definition 1.3.3 (Active ⚙). The set of *active vertices* in configuration $(\vec{e}, h)$ is formally defined as

$$\mathsf{active}(\vec{e}, h) \triangleq \{\mathsf{Thread}(i) \mid e_i \neq ()\} \cup \{\mathsf{Chan}(a) \mid h(a, 0) \neq \perp \lor h(a, 1) \neq \perp\}$$

If $\mathsf{active}(\vec{e}, h) \neq \emptyset$, then there exists a vertex $v \in \mathsf{active}(\vec{e}, h)$ for which we must find a thread that can step. If the vertex v is a thread that can step, we are done. The difficulty is that v may be a thread that is blocked on a $\mathsf{receive}(c)$, where the corresponding buffer of c in heap h is empty. If the configuration is well-formed, then we will presumably be able to find a non-blocked thread connected to the other endpoint of c, since that side will eventually be responsible for sending a message to c. However, the thread holding the other endpoint of c may be blocked itself, waiting on a receive on a different channel. Also, the endpoint c may not even be held by another thread; it could be stored in the buffer of some other channel c'.

Our way out is to use the connectivity graph: starting from vertex v, we search for another thread that can step. To organize this search process, we annotate edges of the connectivity graph with a *waiting direction* (as also done in Section 1.1), depicted as red triangles in Figure 10. The waiting direction is formalized using the notion of being blocked.

Definition 1.3.4 (Blocked ⚙). A vertex v_1 is *blocked* on vertex v_2 in configuration $(\vec{e}, h)$ if v_1 is a thread that is trying to receive from channel v_2 whose buffer is empty. Formally:

$$\mathsf{blocked}_{(\vec{e}, h)}(v_1, v_2) \triangleq \exists i, a, t, K.\ v_1 = \mathsf{Thread}(i) \land v_2 = \mathsf{Chan}(a) \land$$
$$e_i = K[\,\mathsf{receive}(\#(a, t))\,] \land h(a, t) = \epsilon$$

The waiting direction (red triangle) $v_1 \rightarrow_G^l v_2$ of an edge coincides with its ownership direction (black arrowhead) if v_1 is blocked on v_2. Otherwise, it is opposite to the ownership direction.

To find a thread that can step from $v \in \mathsf{active}(\vec{e}, h)$, we follow edges in the waiting direction until we arrive at a vertex that has no outgoing waiting edges. As one can see in Figure 10, if we follow the waiting direction (red triangles) from *any* start vertex v, we always end up in a thread that can step (green dotted square). To prove
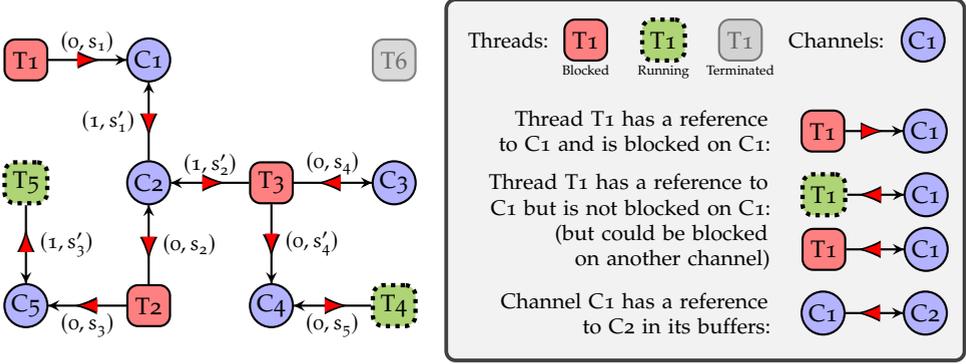
Figure 10: The connectivity graph from Figure 6 annotated with red triangles for the waiting direction.

that we can always find a thread that can step by simply following the waiting edges from any starting vertex, we have to show that:

1. If the current vertex v is a thread, it can either step, or it has an outgoing waiting edge.

2. If the current vertex v is a channel, it always has an outgoing waiting edge.

3. The search process terminates, because the graph is acyclic as an undirected graph.

TO SHOW (1): We show that active threads v can step or have an outgoing waiting arrow by induction on typing. The interesting cases are the channel operations, and receive in particular, so suppose that the thread's expression is $K[\text{receive}(v)]$. By run-time typing, we know that v is a channel reference $\#(a, t)$, and the typing rule for receive(v) gives us the separation logic resource $\text{own}(\{\text{Chan}(a) \mapsto (t, ?\tau.s)\})$. From this it follows that the thread has an outgoing edge to $\text{Chan}(a)$, and hence $\text{Chan}(a)$ has an incoming edge with the label $(t, ?\tau.s)$. From the channel's local well-formedness predicate, it follows that the required buffer exists in the heap. If the buffer is non-empty, then the receive can proceed, so the configuration can step. If the buffer is empty, we have an outgoing waiting arrow from the thread to the channel, so the search process can continue.

TO SHOW (2): The channel v is active, so it has a buffer, so it must have a corresponding incoming edge in the graph by the definition of the local well-formedness predicate for channels. If that incoming edge comes from a vertex v', and that vertex v' is not blocked on v, we are done. That is because then the waiting direction is pointing from v to v', and we can continue the search process from v'. If v' is a thread currently blocked on v, then the session type on that edge must be a receive. It follows from the channel's local well-formedness predicate that

the other endpoint has not yet been closed, and thus there is another incoming edge. It cannot be the case that both buffers are empty and the other edge is also a receive, because that would violate duality. Thus, the other edge is coming from a vertex v'' that is *not* a thread currently blocked on us. So there is a waiting edge from v to v'', and we can continue the search process from v''.

TO SHOW (3): Although (3) is intuitively obvious if one looks at a picture such as Figure 10, one has two difficulties in a formal setting. Firstly, showing that such a search process actually terminates requires formally reasoning about the (undirected) acyclicity of graphs. We refer the interested reader to our appendix and Coq mechanization for details (Jacobs et al., 2021). Secondly, one has to *restructure the argument* in order to even formally state what it means that "the search process terminates". Our key idea is that the progress proof can be proved with an *inductive argument*, with a non-standard graph induction principle.

We call this induction principle for connectivity graphs *waiting induction*. The induction principle says that in order to prove $P(v)$ for all vertices $v \in V$, we can assume that $P(v')$ already holds for all vertices v' that v is waiting for. Note the similarity with strong induction on natural numbers: in order to prove $P(n)$ for all $n \in \mathbb{N}$, we can assume that $P(n')$ already holds for all $n' < n$.

We restructure the progress proof by applying our waiting induction principle at the start. Whenever we said "continue the search process" in the argument above, we can apply the inductive hypothesis. The induction principle is formally stated and discussed in more detail in Section 1.4.

WHAT IS PARAMETRIC IN THIS SECTION. The waiting induction principle is parametric in the types of vertices V and labels L. This induction principle encapsulates the acyclicity reasoning, so that the progress proof can focus on the language-specific reasoning.

1.4 CONNECTIVITY GRAPHS AND WAITING INDUCTION IN DETAIL

Reasoning about graphs in a progress and preservation proof is non-standard, and reasoning about graphs in a proof assistant is more involved than reasoning about inductively-defined types like lists or maps that are normally used to define heap typings. We therefore factor graph reasoning out into a connectivity graph *library* that is *parametric* in vertices V and labels L. In this section we explain the foundations of this library by presenting the formal definition of acyclicity, a selected set of primitive rules (which are used to prove soundness of our separation-logic based graph transformations in Section 1.5 for proving preservation), as well as our principle of waiting induction (for proving progress). We hope to convince the reader that our graph-based approach is feasible—even in a mechanized setting in a proof assistant.

Recall the informal definition of connectivity graphs Cgraph from Section 1.3.2:

$$\text{Cgraph}(V, L) \triangleq \big\{ G \in V \times V \xrightarrow{\text{fin}} L \,\big|\, G \text{ has no undirected cycles} \big\}$$

To define "G has no undirected cycles" formally, we need to introduce some basic notions about graphs. We let $\text{graph}(V, L) \triangleq V \times V \xrightarrow{\text{fin}} L$ be graphs without the acyclicity restriction. The notation $v_1 \rightarrow_G^l v_2$ expresses that there is an edge from vertex v_1 to v_2 with label l (*i.e.*, we have $G(v_1, v_2) = l$). The notation $v_1 \leftrightarrow_G v_2$ expresses that there is an edge from v_1 to v_2 or from v_2 to v_1. The notation $v_1 \leftrightarrow_G^* v_2$ expresses that vertices v_1 and v_2 are *connected* by a (possibly empty) path from v_1 to v_2 where we may follow edges in either direction, and $v_1 \not\leftrightarrow_G^* v_2$ expresses that there is no such path.

Definition 1.4.1 (Undirected acyclicity ✿). A graph $G \in \text{graph}(V, L)$ has no undirected cycles if:

1. The undirected erasure $\bar{G} = \{\{v_1, v_2\} \mid v_1 \leftrightarrow_G v_2\}$, where we forget the labels and directions of the edges, is acyclic. See Jacobs et al. (2021) for details about the formalization of acyclicity of undirected graphs and the undirected erasure.

2. There are no short loops, *i.e.*, we do not both have $v_1 \rightarrow_G^l v_2$ and $v_2 \rightarrow_G^{l'} v_1$.

Our reasoning about the acyclicity of graphs rests on two primitive lemmas:

Lemma 1.4.2 (Graph insertion ✿). *If $G \in \text{graph}(V, L)$ is a graph with no undirected cycles and $v_1 \not\leftrightarrow_G^* v_2$, then $G \cup \{v_1 \rightarrow^l v_2\}$ has no undirected cycles.*

Lemma 1.4.3 (Graph deletion ✿). *If $G \in \text{graph}(V, L)$ is a graph with no undirected cycles and $v_1 \rightarrow_G^l v_2$, then $v_1 \not\leftrightarrow_H^* v_2$ in the graph $H \triangleq G \setminus \{v_1 \rightarrow^l v_2\}$.*

We build a library of derived lemmas on top of these two primitive lemmas. A lemma that is crucial for proving the correctness of our separation logic rules in Section 1.5 is the exchange lemma, which is used to exchange separation logic resources between vertices of the graph:

Lemma 1.4.4 (Graph exchange ✿). *Let $G, H \in \text{graph}(V, L)$ be graphs and let $v_1, v_2 \in V$ be vertices. If*

1. *G has no undirected cycles,*

2. $v_1 \not\leftrightarrow_G^* v_2$,

3. $\text{out}(G, v_1) \uplus \text{out}(G, v_2) = \text{out}(H, v_1) \uplus \text{out}(H, v_2)$, *and*

4. $\text{out}(G, v) = \text{out}(H, v)$ *for all $v \in V \setminus \{v_1, v_2\}$.*

Then:

1. *H has no undirected cycles,*

2. $v_1 \not\leftrightarrow^*_H v_2$, and

3. $\text{in}(G, v) = \text{in}(H, v)$ for all $v \in V$.

This lemma is quite a mouthful, so let us go over it step by step. We start with a graph G and we want to exchange outgoing edges between two unconnected vertices v_1 and v_2 to obtain a graph H in which the union of the outgoing edges of v_1 and v_2 stays the same. The lemma tells us that this operation maintains undirected acyclicity and that v_1 and v_2 are unconnected. Furthermore, the labels of incoming edges stay the same for all vertices.

Note that this property only holds because $\text{in}(G, v)$ is a multiset rather than a map that stores the vertices, like we did for $\text{out}(G, v)$. The fact that the local invariants are unaware of the vertices of origin of the incoming edges is what enables local reasoning: exchange of edges only affects the local invariants of v_1 and v_2. In particular, for a channel it does not matter if its owner changes due to an exchange of resources, because it only matters at which type the channel is owned.

A typical pattern is to compose the lemma for exchange with with lemma for insertion and deletion. For instance, given an edge $v_1 \rightarrow^l_G v_2$, we can first delete the edge using Theorem 1.4.2 to obtain $v_1 \not\leftrightarrow^*_H v_2$. Then we can apply Theorem 1.4.4 to exchange some of the outgoing edges of v_1 and v_2, and then we can re-insert a new edge $v_1 \rightarrow^{l'} v_2$ with a new label l using Theorem 1.4.2.

The lemmas for insertion and deletion (Theorems 1.4.2 and 1.4.3) can not only be used to prove the acyclicity of modified connectivity graphs, but also to prove structural properties of connectivity graphs. The simplest example is a lemma that connectivity graphs have no self loops, which we give here as an illustration that the lemmas for insertion and deletion suffice.[7]

Lemma 1.4.5 (No self loops ⚙). *A connectivity graph* $G \in \text{Cgraph}(V, L)$ *has no self loops* $v \rightarrow^l_G v$.

Proof. Suppose that $v \rightarrow^l_G v$. By Theorem 1.4.3, $v \not\leftrightarrow^*_{G'} v$ in the connectivity graph $G' \triangleq G \setminus \{v \rightarrow^l v\}$. Since every vertex is connected to itself (by definition), we have a contradiction. □

Another example of a structural property that follows from the lemmas for insertion and deletion is the separation lemma. In Section 1.5 this lemma will play an important role in enabling our use of separation logic, where the separating conjunction requires that resources are disjoint.

Lemma 1.4.6 (Separation ⚙). *If* $G \in \text{Cgraph}(V, L)$ *and* $v_1 \not\leftrightarrow^*_G v_2$ *or* $v_1 \leftrightarrow_G v_2$, *then the outgoing edges of* v_1 *and* v_2 *are disjoint, i.e.,* $\text{dom}(\text{out}(G, v_1)) \cap \text{dom}(\text{out}(G, v_2)) = \emptyset$.

Lastly, we have our generic principle of waiting induction that is key to our progress proof.

---

[7] We do actually need this lemma at various points in the Coq proofs of the lemmas in Section 1.5.

Lemma 1.4.7 (Waiting induction ⚙). *Let* $G \in \mathsf{Cgraph}(V, L)$ *be a connectivity graph,* $P \in V \to \mathsf{Prop}$ *a predicate over* V, *and* $R : V \times V \to \mathsf{Prop}$ *a binary relation over* V.
Then in order to prove $\forall v \in V. P(v)$ *it suffices to prove:*

$$\forall v \in V. \left(\forall v' \in V. (v \to_G^l v' \wedge R(v, v')) \Rightarrow P(v') \right) \Rightarrow$$
$$\left(\forall v' \in V. (v' \to_G^l v \wedge \neg R(v', v)) \Rightarrow P(v') \right) \Rightarrow P(v)$$

In other words, to prove $P(v)$, we can assume that $P(v')$ already holds for outgoing neighbors v' of v that are in relation $R(v, v')$, and we can also assume that $P(v')$ holds for incoming neighbors v' of v that are not in relation $R(v', v)$.

Thus, for neighbors $v \to^l v'$, either the proof of $P(v)$ can assume $P(v')$, or *vice versa*, but not both, and the relation $R(v, v')$ determines which. This induction principle is well founded due to the acyclicity of connectivity graphs. We prove this lemma using a similar lemma for undirected graphs, which we detail in Jacobs et al. (2021).

We call the lemma *waiting induction* because (1) we choose $R \triangleq \mathsf{blocked}_{(\vec{e}, h)}$ from Section 1.3.6 and thus R is the waiting relation in our application, and (2) because of the similarity to induction on natural numbers: in order to prove $P(n)$ we can assume that $P(n-1)$ already holds, if $n \neq 0$.

1.5   LOCAL GRAPH TRANSFORMATION RULES IN SEPARATION LOGIC

We now generalize the well-formedness predicate wf from Section 1.3.4 to become parametric in the vertices V and labels L, which involves making it parametric in the local well-formedness predicate to abstract from language-specific aspects. We state separation-logic based rules for the parametric well-formedness predicate so that preservation can be proved using local reasoning. After an initial attempt at a monolithic proof of preservation, we found our approach of separating the graph reasoning from the local language-specific reasoning to be indispensable for mechanization.

Given a local well-formedness predicate $P : V \times \mathsf{Multiset}\, L \to \mathsf{iProp}$, we define the generic global well-formedness predicate $\mathsf{wf}(P)$ as follows:

$$\mathsf{wf}(P) \triangleq \exists G : \mathsf{Cgraph}(V, L). \forall v \in V. P(v, \mathsf{in}(G, v))(\mathsf{out}(G, v))$$

We can instantiate the above definition with $P \triangleq \mathsf{wf}^{local}_{(\vec{e}, h)}$ to obtain the well-formedness predicate from Section 1.3.4 that was tied to our concrete language.

Recall from Section 1.3.5 that preservation means: if $(\vec{e}, h) \rightsquigarrow_{\mathsf{global}} (\vec{e}', h')$, then $\mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e}, h)})$ implies $\mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e}', h')})$. We now present a set of *graph transformation rules* for proving results "$\mathsf{wf}(P)$ implies $\mathsf{wf}(P')$" where P and P' are arbitrary local well-formedness predicates, instead of a concrete local well-formedness predicate. These graph transformation rules perform a transformation of the graph under the hood, but the graphs are encapsulated by the definition of wf, and the rules thus do not

mention any graphs. Instead, the premises of these graph transformation rules ask the user of the rule to prove *local* separation logic entailments involving P and P′.

The first of these graph transformation rules allows the user to exchange separation logic resources between two vertices $v_1, v_2 \in V$ in order to prove that $\mathsf{wf}(P)$ implies $\mathsf{wf}(P')$:

Lemma 1.5.1 (Exchange ✿). *Let $v_1, v_2 \in V$. To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*[8]

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all* $v \in V \setminus \{v_1, v_2\}$ *and* $\Delta \in$ Multiset L

2. $P(v_1, \Delta_1) \twoheadrightarrow \exists l.\ \mathsf{own}(v_2 \mapsto l) * \forall \Delta_2 \in$ Multiset L.
   $(P(v_2, \{l\} \uplus \Delta_2) \twoheadrightarrow \exists l'.\ (\mathsf{own}(v_2 \mapsto l') \twoheadrightarrow P'(v_1, \Delta_1)) * P'(v_2, \{l'\} \uplus \Delta_2))$
   $\qquad$ *for all* $\Delta_1 \in$ Multiset L

This rule generalizes the transformations for send and receive from Figure 9 where resources are exchanged between two vertices. We go over the premises of the rule in detail:

1. The first premise asks the user of the rule to prove the local implication $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ for the vertices $v \in V \setminus \{v_1, v_2\}$ that are not involved in the exchange.

2. The second premise first gives the user access to the local resources $P(v_1, \Delta_1)$ of vertex v_1. The rule then asks the user to prove that there exists an edge $v_1 \rightarrow_G^l v_2$, by showing that $\exists l.\ \mathsf{own}(v_2 \mapsto l) * \cdots$ follows from the local resources of v_1. The rule then gives the user access to the local resources $P(v_2, \{l\} \uplus \Delta_2)$ of vertex v_2, where we have obtained the information that the label $\{l\}$ is part of the incoming edge label multiset of v_2. The rule then allows the user to pick a new label l' for the edge $v_1 \rightarrow^{l'} v_2$. Subsequently, the user has to restore the local resources of v_1 and v_2 for the new P′. For restoring the local resources $P'(v_1, \Delta_1)$, the user additionally gets the $\mathsf{own}(v_2 \mapsto l')$ of the new edge. For restoring the local resources $P'(v_2, \{l'\} \uplus \Delta_2)$, we get the new label in the incoming edge label multiset.

It may seem like this rule only allows us to change the label on the edge $v_1 \rightarrow_G^l v_2$ from l to l', but the rule in fact allows us to arbitrarily exchange separation logic resources (*i.e.*, outgoing edges) between v_1 and v_2. The way this works is that the rule gives us access to the old local resources of both v_1 and v_2, and it asks us to prove the separating conjunction of the new local resources of both v_1 and v_2. The proof rules of separation logic allow us to use resources stored in the old local resources of v_1 to prove the new local resources of v_2, and *vice versa*. Thus, the graph transformation that is applied internally in the rule depends on *which* proof of the separation logic entailment is provided by the user of the rule.

---

[8] Recall that proving $P \in$ iProp means proving $P(\emptyset)$ (see Section 1.3.3), but in practice (and in Coq) this is done using the proof rules of separation logic.

A NOTE ON THE PROOF OF THE TRANSFORMATION RULE. That the transformation rule is able to offer us access to both local resources simultaneously relies crucially on the acyclicity of the graph. The acyclicity, and the existence of an edge between the two vertices, is what allows us to apply the separation lemma (Theorem 1.4.6) that allows us to construct the separating conjunction of the two local resources. In the proof of the rule we re-establish the validity of the resources and the acyclicity of the graph using the exchange lemma (Theorem 1.4.4).

In addition to the preceding transformation rule for changing the label on an edge (and exchanging resources), we have a transformation rule for removing an edge (after exchanging resources). This rule is used in the close case of the preservation proof:

Lemma 1.5.2 (Deallocation ✿). *Let $v_1, v_2 \in V$. To prove* wf(P) *implies* wf(P′), *it suffices to prove:*

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all $v \in V \setminus \{v_1, v_2\}$ and $\Delta \in$ Multiset L*

2. $P(v_1, \Delta_1) \twoheadrightarrow \exists l.\, \text{own}(v_2 \mapsto l) * \forall \Delta_2 \in$ Multiset L.
   $(P(v_2, \{l\} \uplus \Delta_2) \twoheadrightarrow P'(v_1, \Delta_1) * P'(v_2, \Delta_2))$ *for all $\Delta_1 \in$ Multiset L*

We have the following two transformation rules for inserting an outgoing/incoming edge between v_1 and v_2, respectively. To maintain acyclicity, we have to show that v_2 has no existing incoming or outgoing edges. Like the preceding rules, these rules also allow us to transfer resources to v_2. The first lemma below adds an edge $v_1 \rightarrow v_2$ and the second lemma adds an edge $v_2 \rightarrow v_1$.

Lemma 1.5.3 (Allocation out ✿). *Let $v_1, v_2 \in V$. To prove* wf(P) *implies* wf(P′), *it suffices to prove:*

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all $v \in V \setminus \{v_1, v_2\}$ and $\Delta \in$ Multiset L*

2. $P(v_2, \Delta_2) \twoheadrightarrow \ulcorner \Delta_2 = \emptyset \urcorner$ *for all $\Delta_2 \in$ Multiset L*

3. $P(v_1, \Delta_1) \twoheadrightarrow \exists l'.\, (\text{own}(v_2 \mapsto l') \twoheadrightarrow P'(v_1, \Delta_1)) * P'(v_2, \{l'\})$ *for all $\Delta_1 \in$ Multiset L*

Lemma 1.5.4 (Allocation in ✿). *Let $v_1, v_2 \in V$. To prove* wf(P) *implies* wf(P′), *it suffices to prove:*

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all $v \in V \setminus \{v_1, v_2\}$ and $\Delta \in$ Multiset L*

2. $P(v_2, \Delta_2) \twoheadrightarrow \ulcorner \Delta_2 = \emptyset \urcorner$ *for all $\Delta_2 \in$ Multiset L*

3. $P(v_1, \Delta_1) \twoheadrightarrow \exists l'.\, P'(v_1, \Delta_1 \uplus \{l'\}) * (\text{own}(v_1 \mapsto l') \twoheadrightarrow P'(v_2, \emptyset))$ *for all $\Delta_1 \in$ Multiset L*

Lastly, we have a derived transformation rule that adds two new edges $v_1 \rightarrow^{l'_1} v_2$ and $v_2 \leftarrow^{l'_2} v_3$. We use this rule in the fork case of the preservation proof.

Lemma 1.5.5 (Allocation out and in ⚙). *Let* $v_1, v_2, v_3 \in V$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all* $v \in V \setminus \{v_1, v_2, v_3\}$ *and* $\Delta \in \mathsf{Multiset}\ L$

2. $P(v, \Delta) \twoheadrightarrow \ulcorner \Delta = \emptyset \urcorner$ *for all* $v \in \{v_2, v_3\}$ *and* $\Delta \in \mathsf{Multiset}\ L$

3. $P(v_1, \Delta_1) \twoheadrightarrow \exists l_1', l_2'. (\mathsf{own}(v_2 \mapsto l_1') \twoheadrightarrow P'(v_1, \Delta_1 \uplus \{l'\})) * P'(v_2, \{l_1', l_2'\}) *$
   $$(\mathsf{own}(v_2 \mapsto l_2') \twoheadrightarrow P'(v_3, \emptyset))$$
   for all $\Delta_1 \in \mathsf{Multiset}\ L$

This rule can be proved by applying both allocation out and allocation in. It pays off to prove this in the generic setting, because the intermediate state (in which the channel has been allocated but not yet the thread that will hold the other endpoint) is not well-formed according to our wf^{local}. Instead, we prove $\mathsf{wf}(P) \implies \mathsf{wf}(P')$ by carefully choosing Q and proving $\mathsf{wf}(P) \implies \mathsf{wf}(Q)$ using Theorem 1.5.3, and $\mathsf{wf}(Q) \implies \mathsf{wf}(P')$ using Theorem 1.5.4.

1.6 EXTENSIONS

The programming language for which we have mechanized deadlock and memory leak freedom in Coq (Jacobs et al., 2021) supports more features than described in Section 1.2. First, it has more standard features such as sum types, which we do not describe because their rules are standard and the modification of the proof is straightforward. Second, it has unrestricted (non-linear) types, including unrestricted products and sums, and an unrestricted function type (Section 1.6.1) and general equi-recursive functional types (which can encode algebraic data types) and equi-recursive recursive session types (which can encode infinite protocols) (Section 1.6.2). Furthermore, we prove a deadlock freedom property that is stronger than global progress and also rules out partial deadlock (Section 1.6.3).

1.6.1 *Unrestricted Types*

We make the types used for conventional functional programming (such as product, sum, and function types) unrestricted (*i.e.,* non-linear) if their components are unrestricted. Instead of introducing separate linear and non-linear products and sums, we introduce the judgment "τ unrestricted" on types τ, which holds if all the components of τ are unrestricted. Formally, the base types **0**, **1** and **N** are unrestricted, and $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are unrestricted if τ_1 and τ_2 are unrestricted. The type $\tau_1 \multimap \tau_2$ is always linear (*i.e.,* not unrestricted), even if τ_1 and τ_2 are unrestricted, because the closure may capture linear variables. We introduce the type $\tau_1 \rightarrow \tau_2$ of unrestricted functions, which is always unrestricted (even if τ_1 and τ_2 are linear), and whose closures are only allowed to capture unrestricted variables. Selected typing rules are shown in Figure 11. The typing rules involve the *disjointness*

$$\frac{\Gamma \text{ unrestricted}}{\{x \mapsto \tau\} \uplus \Gamma \vdash x : \tau} \qquad \frac{n \in \mathbb{N} \qquad \Gamma \text{ unrestricted}}{\Gamma \vdash n : \mathbf{N}}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \cup \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2 \qquad \Gamma \text{ unrestricted}}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \cup \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

Figure 11: Selected typing rules for unrestricted types.

relation $\Gamma_1 \perp \Gamma_2$, which expresses that Γ_1 and Γ_2 might share unrestricted variables, but otherwise do not overlap. Formally:

$$\Gamma_1 \perp \Gamma_2 \triangleq \forall x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2).\ \Gamma_1(x) = \Gamma_2(x) \wedge \Gamma_1(x) \text{ unrestricted}$$

CHANGES TO THE PROOF. In order to reason about unrestricted values in the separation logic, we add a standard box modality $\square\, P$, defined as $(\square\, P)(\Sigma) \triangleq P(\emptyset) \wedge \Sigma = \emptyset$. The box modality asserts that P does not use any linear resources, which allows it to support proof rules for deletion ($\square\, P \twoheadrightarrow \mathsf{Emp}$) and duplication ($\square\, P \twoheadrightarrow \square\, P * \square\, P$). Lastly, we have the rule $\square\, P \twoheadrightarrow P$ to open the box. We use the box modality in the run-time typing rule for the unrestricted function type:

$$\frac{\square\, (\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2) \quad * \quad \ulcorner \Gamma \text{ unrestricted} \urcorner}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}*$$

The box modality makes sure that the closure cannot capture any channels at run-time.

We prove $(\Gamma \vdash e : \tau) \twoheadrightarrow \square\, (\Gamma \vdash e : \tau)$ if τ unrestricted. This entailment says that run-time typing judgments for expressions e of unrestricted type τ can be freely deleted and duplicated in the separation logic sense. This is crucial for the main change to our proof—the substitution lemma—in which we now have to consider the case that the type is unrestricted, and that a variable could be substituted in multiple or zero places. We use the preceding entailment and the laws of the box modality to adapt the proof of the substitution lemma.

1.6.2 *Equi-Recursive Types*

We extend our type system with equi-recursive functional ($\mu\alpha.\tau$) and session types ($\mu\alpha.s$), in order to be able to encode algebraic data types and infinite protocols,

respectively. We extend the type system with the following rule for unfolding recursive types:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash e : \tau_2}$$

The congruence relation ($\equiv$) relates types up to unfolding of $\mu\alpha.\tau \equiv \tau[\mu\alpha.\tau/\alpha]$. Our mechanization (Section 1.7) is somewhat more general: we use a coinductive definition of types to allow mutual recursion and recursion through the message type as well the tail. We also extend unrestricted types to allow recursive types to be unrestricted. We can encode algebraic data types such as lists by using sums and products and recursive types.

CHANGES TO THE PROOF. We do *not* add a rule for unfolding recursive types to the run-time type system. Rather, we define the run-time type system in a syntax directed way so that all constructors respect the congruence relation ($\equiv$), and then *prove* a version of the unfolding rule:

Lemma 1.6.1. *If $\Gamma_1 \equiv \Gamma_2$ and $\tau_1 \equiv \tau_2$, then $(\Gamma_1 \vdash e : \tau_1) \twoheadrightarrow (\Gamma_2 \vdash e : \tau_2)$.*

EXAMPLE. The combination of equi-recursive and unrestricted types allows us to type check the call-by-value Y-combinator for constructing recursive functions of type $\tau_1 \to \tau_2$. Defining recursive functions in terms of a self-referential type is standard (Harper, 2016):

$$Y : ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)) \to (\tau_1 \to \tau_2)$$
$$Y \triangleq \lambda f. (\lambda x. f (\lambda y. x \, x \, y)) (\lambda x. f (\lambda y. x \, x \, y))$$

We use the recursive type $\mu\alpha.(\alpha \to (\tau_1 \to \tau_2))$ for x and the type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$ for f. Note that while τ_1 and τ_2 can be restricted (linear) types, we *must* use an unrestricted function type for f and x in order to type check the multiple uses of f and x. In fact, a fixed-point combinator for constructing functions $\tau_1 \multimap \tau_2$ with linear function type would violate type safety. Intuitively, a recursive function is allowed to *manipulate* both linear and non-linear resources, but the *definition* of a recursive function is not allowed to *capture* linear resources in its closure because this closure will be invoked multiple times.

1.6.3 *Partial Deadlock and Memory Leak Freedom via Reachability*

In the context of session-typed languages with non-termination (*e.g.,* due to recursive types), deadlock freedom is typically stated as *global progress*, which we prove in Section 1.3.6. Global progress guarantees that the configuration can either take a step, or is in a final state where all threads have successfully terminated and

all channels have been deallocated. Although global progress rules out whole-program deadlocks, as well as memory leaks when all threads have terminated, it admits partial deadlocks as long there is still a thread that can step (*e.g.,* is in an infinite loop). Linear session types actually rule out partial deadlocks and memory leaks even when some threads are still running. Although deadlock freedom and memory leak freedom may seem like separate properties, we state two properties that simultaneously generalize both, namely *partial deadlock/leak freedom* (Theorem 1.6.4) and *full reachability* (Theorem 1.6.6). We prove that these properties are equivalent (Theorem 1.6.7) and show that full reachability can be proven using the waiting induction principle of our proof method (Theorem 1.6.8). Finally, we show that they imply global progress.

In order to arrive at a simultaneous generalization of deadlock and memory leak freedom, consider pure memory leaks and pure deadlocks:

- A *pure memory leak* is one in which we have a set S of channels, such that all endpoints of the channels in S are held by the buffers of channels in the same set S.

- A *pure deadlock* is a set S of both threads and channels with empty buffers, such that all threads in S are blocked on one of the channels in the set S, and all of the endpoints of the channels in S are held by threads in the set S.

In general, we can have a mixed partial deadlock/leak situation that is neither a pure memory leak nor a pure deadlock. Intuitively, a partial deadlock and memory leak is a set S of threads and channels such that all threads in S are blocked on one of the channels in S, and all endpoints of channels in S are held by threads and channels in S. To make this formal, we define the set of vertices $\mathsf{refs}_{(\vec{e},h)}(v) \subseteq V$ that a vertex v references.

Definition 1.6.2. ⚙

$$\mathsf{refs}_{(\vec{e},h)}(\mathsf{Thread}(i)) \triangleq \{\mathsf{Chan}(a') \mid \text{channel } \#(a',t) \text{ occurs in } e_i\}$$
$$\mathsf{refs}_{(\vec{e},h)}(\mathsf{Chan}(a)) \triangleq \{\mathsf{Chan}(a') \mid \text{channel } \#(a',t) \text{ occurs in } h(\#(a,o)) \text{ or } h(\#(a,1))\}$$

With this function at hand, we can define partial deadlock and memory leak freedom.

Definition 1.6.3 (Partial deadlock/leak ⚙). Given a configuration $(\vec{e},h)$, a subset $S \subseteq V$ of the threads and channels is in a partial *deadlock/leak* if the following conditions hold:

1. We have $\emptyset \subset S \subseteq \mathsf{active}(\vec{e},h)$ (see Theorem 1.3.3 for the definition of active).

2. For all threads $\mathsf{Thread}(i) \in S$, the expression e_i cannot step in the heap h.

3. If $\mathsf{Thread}(i) \in S$ and $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Chan}(a))$, then $\mathsf{Chan}(a) \in S$ (see Theorem 1.3.4 for the definition of blocked).

4. If $\mathsf{Chan}(a) \in S$ and $\mathsf{Chan}(a) \in \mathsf{refs}_{(\vec{e},h)}(v)$, then $v \in S$.

Definition 1.6.4 (Partial deadlock/leak freedom ✿). A configuration $(\vec{e}, h)$ is *deadlock/leak free* if no $S \subseteq V$ is in a partial deadlock/leak in $(\vec{e}, h)$.

In order to prove that well-formed configurations have no partial deadlock/leak, we prove another property that we call *full reachability*, which we show to be equivalent to partial deadlock/leak freedom. Full reachability has the advantage that it can be proved directly using waiting induction. It takes inspiration from the notion of reachability used in garbage collection and memory management, namely that data is said to be reachable if it can be reached by transitively following pointers, starting from any thread's stack frames. Memory leak freedom can then be stated as: all data in the configuration is reachable, *i.e.,* there is never any leaked memory. To incorporate deadlock freedom into this, we strengthen the definition of reachability to only start from stack frames of *threads that can step*. However, if a thread T1 is blocked on channel C, and the other endpoint of C is held by still running thread T2, then data held by T1 should also be considered transitively reachable: even though this data is held by a thread that (currently) cannot step, further interaction of T2 with the channel C may unblock T1. We formalize this using the following inductive definition:

Definition 1.6.5 (Reachability ✿). We inductively define the vertices that are *reachable* in $(\vec{e}, h)$:

1. $\mathsf{Thread}(i)$ is reachable if either
   - the expression e_i can step in the heap h, or
   - there exists an a such that $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Chan}(a))$ and $\mathsf{Chan}(a)$ is reachable.

2. $\mathsf{Chan}(a)$ is reachable if there exists a reachable v such that $\mathsf{Chan}(a) \in \mathsf{refs}_{(\vec{e},h)}(v)$.

It is important that reachability is an inductive definition—a coinductive definition would trivially consider all cycles to be reachable.

Definition 1.6.6 (Full reachability ✿). A configuration $(\vec{e}, h)$ is *fully reachable* if all $v \in \mathsf{active}(\vec{e}, h)$ are reachable in $(\vec{e}, h)$.

We show equivalence of partial deadlock/leak freedom and full reachability:

Theorem 1.6.7. ✿ *A configuration $(\vec{e}, h)$ is deadlock/leak free if and only if it is fully reachable.*

For ($\Rightarrow$), we show that none of the objects in a deadlock/leak are reachable, and for ($\Leftarrow$), we show that the set of all non-reachable objects is a deadlock/leak.

Theorem 1.6.8 (Full reachability ✿). *If* $\mathsf{wf}(\vec{e}, h)$, *then* $(\vec{e}, h)$ *is fully reachable.*

This proof goes by waiting induction with $R \triangleq \text{blocked}_{(\vec{e},h)}$ and closely resembles the global progress proof in Section 1.3.6. By using the equivalence between full reachability and partial deadlock/leak freedom, we also obtain that a partial deadlock/leak cannot occur, and can re-prove global progress using reachability.

Corollary 1.6.9 (Partial deadlock/leak freedom ✿). *If* $\text{wf}(\vec{e},h)$, *then* $(\vec{e},h)$ *is deadlock/leak free.*

Corollary 1.6.10 (Global progress' ✿). *If* $\text{wf}(\vec{e},h)$ *and* $\text{active}(\vec{e},h) \neq \emptyset$, *then* $(\vec{e},h)$ *can step.*

The proof of Theorem 1.6.10 uses Theorem 1.6.8, which gives that active objects are reachable. We then find a thread that can step by straightforward induction on the reachability predicate. Alternatively, we can go via Theorem 1.6.9: if none of the threads can step, then the set of all active threads and channels is a deadlock/leak.

Combined with the proofs that the initial configuration ρ of well-typed program satisfies $\text{wf}(\rho)$, and that $\text{wf}(\rho)$ is preserved by the operational semantics (Section 1.3.5), we obtain partial deadlock/leak freedom, full reachability, and global progress for any well-typed program.

1.7 MECHANIZATION IN COQ

Using the Coq proof assistant (Coq Team, 2021) we have mechanized the generic connectivity graph method and its concrete instantiation to our session-typed language. Our mechanization starts with a library for undirected graphs and their acyclicity described in Jacobs et al. (2021). On top of this, we build a library for connectivity graphs and waiting induction (Section 1.4). We combine connectivity graphs with separation logic (Section 1.3.3) to define the generic well-formedness predicate and the separation logic local transformation lemmas (Section 1.5). We instantiate our library by formalizing the language from Section 1.2 with its extensions from Section 1.6. This involves defining the syntax, type system, and operational semantics. For the language-specific parts of our deadlock and leak freedom proof, we define the run-time type system (Section 1.3.3) and the local well-formedness condition (Section 1.3.4). We then prove preservation using our local transformation rules in separation logic (Section 1.3.5), and progress using our principle of waiting induction (Section 1.3.6). We have also mechanized all the extensions (Section 1.6), including unrestricted types (Section 1.6.1), equi-recursive types (Section 1.6.2), and the theorems about reachability and partial deadlock/leak freedom (Section 1.6.3).

LINE COUNTS   The parametric connectivity graph library is 4999 LOC, the language definition is 451 LOC, and the language-specific deadlock and leak freedom proofs are 1688 LOC.

EXTERNAL DEPENDENCIES AND COQ FEATURES THAT WE USE   We use the std++ extended standard library for its results on data structures like lists and finite maps (Coq-std++ Team, 2021). We use the Iris Proof Mode for tactics-based separation logic proofs (Krebbers et al., 2017b, 2018). To represent recursive types (Section 1.6.2), we use the technique by Gay et al. (2020) based on coinductive types combined with Coq's generalized rewriting mechanism to reason up to the congruence ≡ (Sozeau, 2009).

1.8 RELATED WORK

SESSION TYPES   The line of works most closely related to ours are derivatives of Wadler (2012)'s GV, a linear functional language with session types inspired by Gay and Vasconcelos (2010). Whereas Gay and Vasconcelos's calculus does not enjoy the property of deadlock freedom, Wadler's GV and its derivatives (Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021) do. For Wadler's GV, deadlock freedom follows from its translation to CP (Classical Processes) Wadler (2012), for which deadlock freedom holds by cut elimination. Lindley and Morris (2015) then concretize the progress statement by introducing the definition of a deadlocked configuration and proving deadlock freedom using a small-step operational semantics. They also give translations between GV and CP and show that both directions of the translation preserve reductions, unlike previous translations from GV to CP. Subsequently, Lindley and Morris (2015)'s GV has been extended to support least and greatest fixed points (Lindley and Morris, 2016c), exceptions (Fowler et al., 2019), and polymorphism (Lindley and Morris, 2017). A recent extension of GV (Fowler et al., 2021) moreover simplifies GV's meta theory by making process equivalence type preserving. The extension adopts the idea of a hypersequent (Avron, 1991) from (Montesi and Peressotti, 2018; Kokke et al., 2019), yielding Hypersequent GV (HGV).

Like the GV derivatives, our language is a functional language with session-typed channels. Our notion of a connectivity graph moreover bears a resemblance to HGV's abstract process structure (APS), introduced to reason about the acyclic forest structure of a process configuration. However, whereas abstract process structures are defined over hyperenvironments and channel names, our connectivity graph is parametric in its vertices, labels, and edges. More importantly, our connectivity graph is at the core of a proof method for deadlock freedom, fully mechanized in Coq, that uses separation logic and is parametric in its key results. Besides these conceptual differences, there are various technical differences between our formalization and GV formalizations, and even among the different GV variants (such as a synchronous versus an asynchronous semantics). Our formalization uses a standard operational semantics whereas many GV variants use structural congruences and binders for channnel names. In our graph, not only the threads but also channels are vertices, and the edges are directed. Since reasoning about syntax up to equivalence (*e.g.,* structural congruence or α-equivalence) is cumbersome in a

proof assistant like Coq, we believe that our operational semantics is better suited for mechanization (and perhaps closer to how these structures are represented on real computers). Orthogonally, we do not tie channel closing to thread termination and allow close everywhere. As a result our language readily accommodates a forest topology without the need for a special connective, such as mix as used by Fowler et al. (2021).

Earlier non-mechanized work has proved deadlock freedom for a π-calculus using a graphical approach (Carbone and Debois, 2010). This is the earliest work that we are aware of that describes an explicit connection between deadlock freedom and acyclicity of a graph. Their graphical representation is, however, an undirected graph between processes, whereas our connectivity graphs (when instantiated for our language) are directed graphs between threads and channels. Furthermore, their graphs are unlabeled, whereas our connectivity graphs are labeled with session types.

More distantly, our language is related to Toninho et al. (2013); Toninho (2015)'s language SILL, which embeds session-typed processes into a functional core language via a contextual monad. The language is based on the Curry-Howard correspondence established by between intuitionistic linear logic and session-typed π-calculus. Deadlock freedom of SILL follows thus as a consequence. Due to its modal separation, SILL does not allow mixing of functional and session terms freely, in contrast to GV and our language. The seminal paper by Caires and Pfenning (2010) and Toninho (2015)'s thesis spurred a series of derivatives, similarly to Wadler's CP and GV, accommodating, for example, polymorphism (Caires et al., 2013; Pérez et al., 2014), work analysis (Das et al., 2018), and information flow control (Derakhshan et al., 2021). Due to their connection to intuitionistic linear logic, all these works guarantee deadlock freedom. However, unlike ours, none of these deadlock freedom proofs have been mechanized in a proof assistant.

A derivative of SILL, SILL$_S$ (Balzer and Pfenning, 2017), introduces a controlled form of aliasing through a stratification of linear and shared session types with adjoint modalities (Pfenning and Griffith, 2015; Benton, 1994; Reed, 2009b) to support multiple-client scenarios. Whereas the resulting language reclaims the expressiveness of the untyped asynchronous π-calculus for session-typed languages (Balzer et al., 2018), it also sacrifices deadlock freedom (rectified by its successor SILL$_S^+$ (Balzer et al., 2019)). Recent extensions of classical linear logic session types contribute another approach to softening the rigidity of linear session types to support multiple client sessions and nondeterminism (Qian et al., 2021) and memory cells and nondeterministic updates (Rocha and Caires, 2021), respectively. Whereas neither of these recent approaches reclaim the full expressiveness of unrestricted sharing, they keep the logical foundation intact and thus uphold deadlock freedom. However, none of these works have been mechanized in a proof assistant.

Prior to the development of logic-based session types (Caires and Pfenning, 2010; Wadler, 2012), deadlock freedom in session-typed calculi (Vasconcelos, 2012) was guaranteed only for processes interacting on a *single* session—interleaving of blocking

actions on different sessions could easily result in deadlocks. To address limitations of classical binary session types, Honda et al. (2008) introduced *multiparty* session types, where sessions are described by so-called global types that capture the interactions between an arbitrary number of session participants. Given some well-formedness constraints, global types can ensure that a collection of processes correctly implement the global behavior in a deadlock-free way. However, these global type-based approaches do not ensure deadlock freedom in the presence of higher-order channels, interleaved sessions, dynamic channel creation, or dynamic thread creation. To remedy the deficiency various extensions at increasing degrees of complexity were introduced. For example, Bettini et al. (2008) and Coppo et al. (2016) track usage orders among interleaved multiparty sessions, ruling out cyclic dependencies but also restricting recursion. Our approach instead supports higher-order channels, general recursion, and deadlock freedom solely using a linear type system, by restricting to binary sessions.

SEPARATION LOGIC   Separation logic (O'Hearn et al., 2001) is conventionally used in Hoare-style program logics for proving functional correctness, while we use it to define and reason about (run-time) typing judgments. In conventional separation logic, propositions are predicates over heaps (possibly extended with permissions, ghost state, *etc.*), whereas we consider predicates over the outgoing edges of a connectivity graph (which contain types instead of values). The idea of using separation logic to define typing judgments for linear languages has been explored by Rouvoet et al. (2020, 2021) in the context of intrinsically-typed programming in Agda. They present separation-logic based programming abstractions to hide types of references in intrinsically-typed interpreters, and to hide types of labels in intrinsically-typed compilers. As a case study, Rouvoet et al. (2020) use their abstractions to define an intrinsically-typed interpreter for a small session-typed language that guarantees type safety by construction (but not deadlock or resource leak freedom).

Separation logic has also been used to define logical relation models of affine type systems. For example, logical relations in the Iris separation logic (Jung et al., 2015, 2018b) have been used for proving memory safety and data race freedom of Rust (Jung et al., 2018a), as well as type safety of session types (Hinrichsen et al., 2021). To extend the logical-relations based approach to prove deadlock freedom, a full-fledged separation logic that is capable of proving deadlock freedom is needed. While separation logics and Hoare logics with support for deadlock freedom exist, *e.g.,* (Hamin and Jacobs, 2018; Le et al., 2013; Zhang et al., 2016), they use lock-orders, whose logical expressivity is different from session types. Some separation logics have support for pointed-by assertions (Kassios and Kritikos, 2013), which can be used to reason about memory leak freedom.

Various extensions of separation logic that incorporate session-type based mechanisms to reason about message-passing programs have been developed, *e.g.,* Francalanza et al. (2011); Lozes and Villard (2012); Craciun et al. (2015); Oortwijn

et al. (2016); Hinrichsen et al. (2020, 2021). The goal of these logics is different from ours—they are full-fledged Hoare logics aimed at proving functional correctness instead of deadlock freedom. On the other hand, we use the assertion layer of separation to hide bookkeeping in the definition of run-time typing judgments, and to describe connectivity graph transformations in an abstract and generic way.

MECHANIZED RESULTS OF SESSION TYPES   Thiemann (2019) proves type safety of a linear λ-calculus with session types that is inspired by GV. They do not prove deadlock or memory leak freedom. Their mechanization involves an extensive amount of bookkeeping to keep track of resources. Rouvoet et al. (2020) streamlined this approach via separation logic (see discussion above).

Hinrichsen et al. (2021) prove type safety for a comprehensive session-typed language with locks, subtyping and polymorphism using Iris in Coq. Their type system is affine, which means that deadlocks are considered safe (their receive operation will spin if the buffer is empty). Their proof is based is on logical relations instead of progress and preservation (see discussion above).

Tassarotti et al. (2017) prove correctness of a compiler for an affine session-typed language using Iris in Coq. The operational semantics of their source language is similar to ours, while channels are compiled to an implementation involving linked lists in the target. Their compiler is proved to be termination preserving, so a target program deadlocks iff the source deadlocks.

More distantly, there also exist various mechanized results involving π-calculus. Goto et al. (2016) prove type safety for a π-calculus with a polymorphic session type system in Coq. Their type system allows dropping channels, and hence does not enjoy deadlock nor memory leak freedom. Ciccone and Padovani (2020) mechanize dependent binary session session types by embedding them into a π-calculus in Agda. They prove subject reduction (*i.e.,* preservation) and that typing is preserved by structural congruence. Neither deadlock freedom nor leak freedom is proved. Castro-Perez et al. (2020) present a framework for locally-nameless representations of π-calculus in Coq. They use their framework to prove subject reduction (*i.e.,* preservation) of a type system for binary session types. Neither deadlock freedom nor leak freedom is proved. Their framework is used by Castro-Perez et al. (2021) to mechanize a DSL for multiparty communication in Coq based on asynchronous multiparty session types. They prove deadlock freedom w.r.t. a global type, but do not prove deadlock freedom in the presence of higher-order channels, interleaved sessions, dynamic channel creation, or dynamic thread creation.

Gay et al. (2020) study various notions of duality in Agda, and show that distribution laws for duality over the recursion operator are unsound. Unlike the other mechanized results discussed so far, they focus on the static instead of dynamic semantics of session types. We have adapted their approach of using coinductive types for mechanizing general recursive session types (see Section 1.7). Keizer et al. (2021) use coalgebras to model session types in a non-mechanized setting.

More distantly related are mechanized versions of cut elimination of linear logic (Reed, 2009a; Chaudhuri et al., 2019), which by Curry-Howard relates to deadlock freedom of intuitionistic session types. The authors were incentivized by mistakes in various existing, non-mechanized proofs. However, whereas a cut elimination proof concerns a logical inference system only, our proof of deadlock freedom encompasses a typed programming language with operational semantics, requiring us to reason not only about its statics but also it execution semantics. Moreover, our language includes features such as recursive types (Section 1.6.2) that break cut elimination.

Mechanization results, lastly, also exist for choreographic languages (Montesi, 2021). Cruz-Filipe et al. (2021a) mechanize choreography compilation in Coq for the choreographic language Core Choreographies (CC) introduced by Cruz-Filipe et al. (2021b). CC supports recursion and its semantics has been formalized in Coq by Cruz-Filipe et al. (2021b). Key results of the formalization include determinism, confluence, and deadlock-freedom by design as well as Turing completeness.

PROCESS CALCULI   The addition of channel usage information to types in a concurrent, message-passing setting was pioneered by Kobayashi (1997); Igarashi and Kobayashi (1997); Kobayashi et al. (1999), who applied the idea to deadlock prevention in the π-calculus and later to more general properties (Igarashi and Kobayashi, 2001, 2004), giving rise to a generic system that can be instantiated to produce a variety of concrete typing disciplines for the π-calculus. Typically, types are augmented with the relative ordering of channel actions, with the type system ensuring that the transitive closure of such orderings forms a strict partial order, ensuring deadlock-freedom. Building on this, Kobayashi (2002a) proposed type systems that ensure a stronger property, dubbed lock freedom, and variants that are amenable to type inference (Kobayashi et al., 2000). Kobayashi (2006) extended this to account for recursive processes and type inference. Kobayashi-style systems have also been adopted for session-typed languages (Dardha and Gay, 2018; Balzer et al., 2019).

1.9 FUTURE WORK

We have used our connectivity graph method to give a mechanized proof of deadlock and memory-leak freedom for binary session types. Since connectivity graphs are not restricted to two incoming edges per channel, we would like to explore language designs with a version of multiparty session types that supports dynamic thread and channel creation, and higher order channels, but still enjoys global progress from typing in a manner similar to binary session types (*i.e.*, without additional mechanisms such as orders or priorities). Second, we would like to explore whether other concurrency mechanisms such as locks and barriers could be handled by our method.

Chapter 2

Higher-Order Leak and Deadlock Free Locks

ABSTRACT   Reasoning about concurrent programs is challenging, especially if data is shared among threads. Program correctness can be violated by the presence of data races—whose prevention has been a topic of concern both in research and in practice. The Rust programming language is a prime example, putting the slogan fearless concurrency in practice by not only employing an ownership-based type system for memory management, but also using its type system to enforce mutual exclusion on shared data. Locking, unfortunately, not only comes at the price of *deadlocks* but shared access to data may also cause memory *leaks*.

This chapter develops a theory of deadlock and leak freedom for *higher-order locks* in a shared memory concurrent setting. Higher-order locks allow sharing not only of basic values but also of other locks and channels, and are themselves first-class citizens. The theory is based on the notion of a *sharing topology*, administrating who is permitted to access shared data at what point in the program. The paper first develops higher-order locks for *acyclic* sharing topologies, instantiated in a λ-calculus with higher-order locks and message-passing concurrency. The paper then extends the calculus to support *circular* dependencies with *dynamic* lock orders, which we illustrate with a dynamic version of Dijkstra's dining philosophers problem. Well-typed programs in the resulting calculi are shown to be free of deadlocks and memory leaks, with proofs mechanized in the Coq proof assistant.

2.1   INTRODUCTION

Today's applications are inherently concurrent, necessitating programming languages and constructs that support spawning of threads and *sharing* of resources. Sharing of resources among threads, a sine qua non for many applications, is the source of many concurrency-related software bugs. The issue is the possibility of a race condition, if simultaneous write and read accesses are performed to shared data. To rule out data races, locks can be employed, forcing simultaneous accesses to happen in mutual exclusion from each other. Locking unfortunately not only comes at the cost of *deadlocks*, but shared access to data may also cause *memory leaks*.

This chapter develops a λ-*calculus* with *higher-order locks* and message-passing concurrency, where well-typed programs are free of memory leaks and deadlocks. Whereas there exist type systems for memory safety—most notably Rust (Matsakis and Klock, 2014; Jung et al., 2018a), incorporating ideas of ownership types (Clarke et al., 1998; Müller, 2002) and region management (Tofte and Talpin, 1997; Grossman

et al., 2002)—memory safety only ensures that no dangling pointers are dereferenced, but does not rule out memory leaks. Similarly, type systems for deadlock and leak freedom have been developed, pioneered by Kobayashi (1997); Igarashi and Kobayashi (1997); Kobayashi et al. (1999) in the context of the π-calculus and by Caires and Pfenning (2010); Wadler (2012) in the context of linear logic session types. Our work builds on the latter and extends it with the capability to share resources as present in a shared memory setting.

It may come as a surprise that locking not only can cause deadlocks but also memory leaks. We give an example of a memory leak caused by a mutex in Rust below:

```rust
struct X { x: Option<Arc<Mutex<X>>> } // type that will be stored in the mutex

let m1 = Arc::new(Mutex::new(X { x: None })); // create mutex with empty payload
let m2 = m1.clone(); // a second reference to the mutex, incrementing refcount
let mut g = m1.lock(); // acquire the mutex, giving access to the contents
*g = X { x: Some(m2) }; // mutate the contents to store m2 in the mutex
drop(g); // release the lock
drop(m1); // drop the reference to the mutex, decrementing the refcount
```

On the first line, we declare a recursive struct, that optionally contains a reference to a mutex that is reference counted. On the second line, we then create such a mutex, initially with empty payload. We then clone the reference to the mutex, raising the reference count to 2. Finally, we lock the mutex through the first reference and store the second reference in it, transferring ownership of m2 to the mutex. On the last line, we release the mutex and drop the reference to m1. This decrements the reference count to 1, but there still exists a self-reference from inside the mutex, leading to a memory leak.

It is tempting to conclude from the above example that recursive types are necessary to create memory leaks. This is not the case, however. Instead of storing the mutex inside the mutex directly, one can store a closure of type unit -> unit that captures the mutex in its lexical environment.

Memory leaks can moreover be caused by channels, as illustrated by the below Rust code:

```rust
struct Y { y: Receiver<Y> } // declare type that will be sent over the channel

let (s,r) = mpsc::channel(); // create a channel with sender s and receiver r
s.send(Y { y: r }); // put the receiver in the buffer of the channel
drop(s); // drop the reference to the sender; but memory is leaked
```

On the first line, we declare a recursive struct with a reference to a receiver endpoint of a channel. On the second line, we then allocate a channel, which gives us a sender s and receiver r. We then send the receiver along the sender, transferring it into the channel's buffer. When we drop the sender, the reference to the receiver still exists from within the buffer, creating a memory leak.

Unsurprisingly, we can use the same concurrency constructs to also cause deadlocks. For example, a thread my allocate a new channel, keep both the sender and the receiver reference, and attempt to receive from the receiver before sending along the sender:

```
let (s,r) = mpsc::channel(); // create a new channel
r.recv(); // this call blocks on receiving a message, deadlock!
s.send(3); // the message is sent, but too late
```

Similarly, mutexes can give rise to deadlocks. Consider the following swap function:

```
fn swap(m1: &Mutex<i32>, m2: &Mutex<i32>) {
  let mut g1 = m1.lock(); // acquire first mutex
  let mut g2 = m2.lock(); // acquire second mutex
  let tmp = *g1; // obtain the contents stored in m1
  *g1 = *g2; // replace the contents of m1 with the contents of m2
  *g2 = tmp; // replace the contents of m2 with the original contents of m1
  drop(g1); drop(g2) // release the locks
}
```

This function takes two references to mutexes, locks both, and swaps their contents. Now let's consider the below code that calls this function:

```
let m1 = Arc::new(Mutex::new(1)); // create a new mutex
let m2 = m1.clone(); // create a second reference to the mutex
swap(&m1,&m2); // deadlock!
```

The code allocates a mutex, yielding the reference m1, and then creates an alias m2 to the same mutex. Then it calls function swap with m1 and m2 as arguments. The function will deadlock upon the second acquire, which will block until the first one is released. This last example also demonstrates that reasoning about deadlocks—and for that matter memory leaks—is not inherently local. Both the function swap and the above code are benign on their own but problematic when composed.

The above examples use the API constructs Arc<Mutex<T>> and Rc<RefCell<T>> to cause memory leaks and deadlocks, suggesting that substructural typing is insufficient to rule out memory leaks and deadlocks, but that memory leak and deadlock freedom must be accounted for at the level of API design. Based on this observation, we develop a λ-calculus for shared memory concurrency with a lock data type, guaranteeing absence of *memory leaks* and *deadlocks* by type checking. Memory leaks are especially bothersome for resource-intensive applications, and deadlocks can prevent an entire application from being productive. We phrase our lock API and type system in a λ-calculus setting to keep it independent of an actual target language, yet readily adoptable by any language with similar features. Locks in our calculus are *higher-order*, allowing them to store not only basic values but also other locks. This feature enables us to encode *session typed channels* (Honda, 1993; Honda et al., 1998). These channels are also higher-order, and can thus be stored in locks and sent over each other as well.

While higher-order locks and channels increase the expressivity of our calculus and scale it to realistic application scenarios, they also challenge our goal to ensure deadlock and leak freedom by type checking. Our approach is to account for an application's *sharing topology*, which tracks, for every lock, *(i)* who has references to the lock, *(ii)* who is responsible for releasing the lock, and *(iii)* who is responsible for deallocating the lock. The fundamental invariant that we place on the sharing topology demands that there never exist any circular dependencies among these responsibilities at any point in the execution of a program.

We first develop the calculus λ_{lock}, which enforces this invariant preemptively, by demanding that the sharing topology be *acyclic*. As a result, λ_{lock} enjoys memory leak and deadlock freedom, with corresponding theorems and proofs developed in Section 2.4 and Section 2.5, respectively. We then introduce the calculus $\lambda_{\text{lock++}}$, an extension of λ_{lock} that supports circular resource dependencies, as famously portrayed by Dijkstra's dining philosophers problem, while preserving memory leak and deadlock freedom. $\lambda_{\text{lock++}}$ permits *cyclic* sharing dependencies within *lock groups* using a *lock order*, but satisfies the sharing topology's fundamental invariant between different lock groups. These orders are purely local to a lock group and can change dynamically by the addition or removal of locks to and from a group. Local orders are compositional in that they remove the need for local orders to comply with each other or a global lock order when acquiring locks from distinct groups. The proofs of memory leak and deadlock freedom for λ_{lock} and $\lambda_{\text{lock++}}$ are mechanized in the Coq proof assistant and detailed in Section 2.7.

IN SUMMARY, THIS CHAPTER CONTRIBUTES

- A notion of acyclic sharing topology to rule out circular dependencies without any restriction on the order in which operations must be performed,

- the language λ_{lock} with higher-order locks for shared memory concurrency and type system based on the sharing topology to ensure memory leak freedom and deadlock freedom,

- an encoding of session-typed message-passing channels in terms of locks,

- the language $\lambda_{\text{lock++}}$, an extension supporting cyclic unbounded process networks,

- proofs of deadlock and memory leak freedom for well-typed λ_{lock} and $\lambda_{\text{lock++}}$ programs, mechanized in Coq.

This section develops the notion of a *sharing topology* underlying our calculus and illustrates its type system based on examples. We start by deriving the fundamental invariant to be preserved by the sharing topology in several steps, distilling several key principles. We first focus on *acyclic* sharing topologies, resulting in the calculus λ_{lock}, which we then extend to account for *cyclic* sharing dependencies, resulting in the calculus $\lambda_{\text{lock++}}$.

2.2.1 *Invariant for Leak and Deadlock Freedom*

The examples of memory leaks and deadlocks discussed in Section 2.1 all share a common pattern: a thread has several references to the same lock, introducing self-referential responsibilities for releasing and deallocating locks. **Our goal is to devise a system that allows threads to reason *locally* about shared resources and, in particular, to give threads complete freedom to acquire and release any lock they reference.** Our approach thus opts for restricting the propagation of lock references by an individual thread rather than their use. To forbid the self-referential scenarios discussed in Section 2.1, the fundamental invariant of the sharing topology must satisfy the following principle:

> **Principle 1: Each thread only holds one reference to any given lock.**

To satisfy this principle our calculus treats locks *linearly*, ensuring that references to locks cannot be duplicated *within* a thread.

The above principle is obviously not yet sufficient for ruling out deadlocks, as deadlocks can also result when two threads compete for resources. For example, consider two threads T_1 and T_2 with references to locks l_1 and l_2. A deadlock can arise if thread T_1 tries to acquire lock l_1 and then l_2, and thread T_2 tries to acquire lock l_2 and then lock l_1. Therefore, the fundamental invariant must also satisfy the following principle (which Section 2.6 relaxes by permitting sharing of a group of locks rather than an individual lock):

> **Principle 2: Any two threads may share at most one lock.**

This principle is still not yet sufficient for ruling out deadlocks. Consider an example with 3 threads T_1, T_2, T_3, and 3 locks l_1, l_2, l_3, where:

- thread T_1 acquires l_1 and then l_2
- thread T_2 acquires l_2 and then l_3
- thread T_3 acquires l_3 and then l_1

If a schedule allows each thread to acquire their first lock, the threads will subsequently deadlock when trying to acquire their second lock. Note, however, that the preceding principle is satisfied: thread T_1 and T_2 only share lock l_2, thread

T_2 and T_3 only share lock l_3, and tread T_3 and T_1 only share lock l_1. Thus, if we want to uphold thread local reasoning, while guaranteeing that thread composition preserves deadlock freedom, we must impose a stronger invariant, constraining the sharing topology:

Principle 3: If we consider the graph where threads are connected to the locks they hold a reference to, this graph must not have a cycle.

Our calculus enforces this principle by the following lock and thread operations, which bear a resemblance to channels in linear logic session types based on cut elimination (Caires and Pfenning, 2010; Wadler, 2012):

- **new** to create a new lock. Threads are free to use this operation. Because the lock is created, the creating thread is the only one to have a reference to it.

- **acquire** to acquire a lock. This operation can be called at any time, but the type system must ensure that the same lock cannot be acquired multiple times via the same reference.

- **release** to release the lock. This operation can be called at any time, and the type system *must* ensure that it is called eventually for any acquired lock.

- **fork**, which forks off a new thread, and allows the programmer to create a new reference to *one* lock from the parent thread and share it with the child thread.

Although the **fork** construct allows duplicating a lock reference, the newly created reference must be passed to the forked off thread, creating a new edge between the new thread and the lock. If we restrict sharing of a lock between a parent and child thread to exactly *one* lock, the graph arising from the reference structure between threads and locks remains acyclic. For example, consider the threads T_1, T_2, T_3 and locks l_1 and l_2 such that:

- threads T_1 and T_2 share lock l_1

- threads T_1 and T_3 share lock l_2

If T_1 spawns T_4 and provides a reference to l_1, the resulting reference structure remains acyclic: T_1 is connected to l_1 and l_2, with the former being connected to T_2 and T_4 and the latter being connected to T_3. However, if we allowed T_1 to share both l_1 and l_2 with T_4, the graph becomes cyclic: T_1 is connected to T_4 both via l_1 and l_2.

The type system that we sketch in the next section and detail in Section 2.3 enforces the above rules and thus upholds the principles derived so far to rule out deadlocks. A reader may wonder whether these principles are strong enough for asserting deadlock freedom in the presence of higher-order locks, allowing us to store locks in locks. For example, we can easily transfer a lock l_1 from thread T_1 to thread T_2 by storing it in a lock l_2 shared between T_1 and T_2, allowing T_2 to retrieve l_1 by acquiring l_2. This scenario is indeed possible and turns out not to be a problem.

While not immediately obvious, this transfer actually preserves acyclicity of the sharing topology. To account for the possibility of references between locks, we refine our invariant as follows:

Principle 4: If we consider the graph where threads are connected to the locks they hold a reference to, and locks are connected to locks they hold a reference to, this graph must not have a cycle.

Principle 4 amounts to an invariant that is sufficient to ensure deadlock freedom. Section 2.5 details that a well-typed λ_{lock} program preserves this invariant along transitions. The next question to explore is whether this invariant is also sufficient to ensure memory leak freedom.

It seems that the above invariant is sufficient for ruling out the examples of memory leaks portrayed in Section 2.1, because they are all instances of self-referential structures, prevented by the above invariant. However, to answer this question entirely, we have to remind ourselves of our definition of a sharing topology given in Section 2.1:

Definition 2.2.1. A *sharing topology* tracks, for every lock, *(i)* who has references to the lock, *(ii)* who is responsible for releasing the lock, and *(iii)* who is responsible for deallocating the lock.

So far, we have only accommodated the first two ingredients, but yet have to establish responsibility of lock deallocation.

To get started, let us first explore the question of *"how to ever safely get rid of a lock"*. Obviously, we should not attempt to drop a reference to a lock that we have acquired, because then this lock would never be released, blocking any other threads that are trying to acquire that lock. So, is it then safe to drop a reference to a lock that we have *not* currently acquired? As a matter of fact even this is not safe, if we allow storing linear data in locks. For example, we could then easily discard a linear value v as follows, which would defeat the purpose of linear typing:

1. Create a new lock and acquire it.

2. Put the linear value v in the lock.

3. Release the lock.

4. Drop the reference to the lock.

We thus face the following conundrum: if we allow dropping references to an acquired lock, then we cannot leak data, but we get deadlocks, and if we allow dropping references to a non-acquired lock, then we can leak data (which then allows us to create deadlocks anyway).

It seems that we have to circle back to Theorem 2.2.1 and find a way to designate one reference among all the references to a lock as the one that carries the responsibility for deallocation. For this purpose we differentiate lock references

into an *owning reference*, which carries the responsibility to deallocate the lock, and *client references*, which can be dropped. Naturally, there must exist exactly one owning reference. An owning reference can only be dropped after the lock has been deallocated. To deallocate the lock, the owner must first *wait* for all the clients to drop their references and then retrieve the contents of the lock.

This brings us to our final invariant:

> **Principle 5: If we consider the graph where threads are connected to the locks they hold a reference to, and locks are connected to locks they hold a reference to, this graph must not have a cycle. Furthermore, each lock must have precisely one owning reference, and zero or more client references.**

2.2.2 The $\textbf{Lock}\langle \tau_b^a \rangle$ Data Type and its Operations

Let us now investigate what a lock API and type system based on these principles look like. A detailed discussion of the resulting language λ_{lock} is given in Section 2.3.

We introduce the following type of lock references:

$$\textbf{Lock}\langle \tau_b^a \rangle$$

where

- $\tau \in \textbf{Type}$ is the type of values stored in the lock.

- $a \in \{0, 1\}$ indicates whether we are the owner ($a = 1$) or a client ($a = 0$).

- $b \in \{0, 1\}$ indicates whether we have acquired the lock ($b = 1$) or not ($b = 0$).

λ_{lock} supports the following operations to acquire and release locks:

$$\textbf{acquire} : \textbf{Lock}\langle \tau_0^a \rangle \to \textbf{Lock}\langle \tau_1^a \rangle \times \tau$$
$$\textbf{release} : \textbf{Lock}\langle \tau_1^a \rangle \times \tau \to \textbf{Lock}\langle \tau_0^a \rangle$$

These operations are *linear* and hence consume their argument. Both operations return the lock argument reference at a different type, reflecting whether the lock is currently acquired or not. The acquire operation gives the user full access to the τ value protected by the lock, and the release operation requires the user to put back a τ value. Acquire and release operations work for $a \in \{0, 1\}$, so both clients and the owner are allowed to acquire and release the lock. We find it helpful to think of a lock as a shared "locker" or container to exchange valuables. Using this metaphor, we can perceive an acquire as opening the closed locker to retrieve the valuable and a release as closing an open locker to store the valuable. If a reference ℓ is of type $\textbf{Lock}\langle \tau_1^a \rangle$, indicating that the locker has been opened, the holder of the reference is responsible for eventually putting back the valuable using a **release**. If a reference ℓ is of type $\textbf{Lock}\langle \tau_0^a \rangle$, indicating that the locker has not been opened via the reference ℓ, the holder of the reference is allowed to try to acquire the locker.

Let us now look at how locks are created and destroyed. We have three operations, one for creating a lock, one for deallocating a lock via its owning reference, and one for dropping a client reference to a lock:

$$\mathbf{new} : \mathbf{1} \rightarrow \mathbf{Lock}\langle \tau_1^1 \rangle$$

$$\mathbf{wait} : \mathbf{Lock}\langle \tau_0^1 \rangle \rightarrow \tau$$

$$\mathbf{drop} : \mathbf{Lock}\langle \tau_0^0 \rangle \rightarrow \mathbf{1}$$

The operation **new** creates an owning reference. The operation **wait** on the owning reference waits for all clients to finish and then returns ownership of the value τ stored in the lock (and frees the memory associated with the lock). The operation **drop** on a client reference yields unit, effectively not returning anything. The drop operation could potentially be automatically inserted by a compiler, as it is done by the Rust compiler, for example, but we prefer to be explicit. Note that both **wait** and **drop** require the lock to be in a non-acquired (*a.k.a.,* closed) state, which means that a thread holding an open lock reference must fulfill its obligation to put a value back into the lock using **release** before it is allowed to use **drop** or **wait** on that lock reference. This ensures that **drop** and **wait** cannot cause another thread's **acquire** to deadlock. The details of deadlock freedom can be found in Sections 2.4 and 2.5.

Client references are created upon fork:

$$\mathbf{fork} : \ \mathbf{Lock}\langle \tau_{b_1+b_2}^{a_1+a_2} \rangle \times (\mathbf{Lock}\langle \tau_{b_2}^{a_2} \rangle \multimap \mathbf{1}) \ \rightarrow \ \mathbf{Lock}\langle \tau_{b_1}^{a_1} \rangle$$

It may be helpful to consider an example, where ℓ has type $\mathbf{Lock}\langle \tau_{b_1+b_2}^{a_1+a_2} \rangle$:

$$\mathbf{let} \ \ell_1 : \mathbf{Lock}\langle \tau_{b_1}^{a_1} \rangle \ = \ \mathbf{fork}(\ell, \lambda \ell_2 : \mathbf{Lock}\langle \tau_{b_2}^{a_2} \rangle. \ (\cdots))$$

The fork operation consumes the original lock reference ℓ, and splits it into two references, ℓ_1 and ℓ_2. The reference ℓ_1 is returned to the main thread, and the reference ℓ_2 is passed to the child thread. The child thread runs the code indicated by $(\cdots)$, which has access to ℓ_2. In terms of types,

$$\mathbf{Lock}\langle \tau_{b_1+b_2}^{a_1+a_2} \rangle \quad \text{is split into} \quad \begin{cases} \mathbf{Lock}\langle \tau_{b_1}^{a_1} \rangle \\ \mathbf{Lock}\langle \tau_{b_2}^{a_2} \rangle \end{cases}$$

such that $a_1 + a_2 \leqslant 1$ and $b_1 + b_2 \leqslant 1$. This condition ensures that if the original reference ℓ is an owner reference and thus of type $\mathbf{Lock}\langle \tau_b^1 \rangle$, it can only be split into an owner reference and client reference. Conversely, if the original lock reference ℓ is a client reference and thus of type $\mathbf{Lock}\langle \tau_b^0 \rangle$, it can only be split into two client references. Similarly, if the original reference ℓ is acquired and thus of type $\mathbf{Lock}\langle \tau_1^a \rangle$, only one of the new references is acquired. If the original reference ℓ is not acquired and thus of type $\mathbf{Lock}\langle \tau_0^a \rangle$, the two new references are not acquired either.

The standard rules of binding and scope apply to the lambda used in a **fork** as well. For example, we can transfer linear resources from the main thread to a child thread, *e.g.,* the resource bound to the linear variable r in the following example:

$$\text{let } \ell \ = \ \textbf{new}() \textbf{ in}$$
$$\text{let } r \ = \ \textbf{new}() \textbf{ in}$$
$$\text{let } \ell_1 \ = \ \textbf{fork}(\ell, \lambda\ell_2. \ (\cdots r \cdots)) \textbf{ in } (\cdots)$$

Here, the resources r can no longer be used in the main thread because of linearity.

These are all the constructs of λ_{lock} that concern locks. Section 2.3 details how we integrate λ_{lock} with session-typed channels, facilitating the exchange of locks between threads not only by storing them into other locks, but also by sending them along channels, possibly as part of a compound data structure. Channels, of course, are first-class as well, allowing them to be sent over each other and stored in locks. Given the range of possibilities of how the sharing topology of a program can change dynamically, a reader may be surprised that λ_{lock} asserts memory leak and deadlock freedom by type checking. After all, as usual, the devil is in the details! The formal statement of memory leak and deadlock freedom is given in Section 2.4, and their proof is sketched in Section 2.5. For the full details, the reader is referred to the mechanization Section 2.7.

2.2.3   *Examples*

We now look at a few examples that illustrate the use of lock operations.

Locks as mutable references

A lock without any client references can be viewed as a linear mutable reference cell. We can create such a reference cell with $\ell = \textbf{release}(\textbf{new}(), v)$, read its value with $\textbf{acquire}(\ell)$, and write into it a new value with $\textbf{release}(\ell, v)$. We can also deallocate the reference with $\textbf{wait}(\ell)$, which gives us back the value.

$$\text{let } \ell \ = \ \textbf{release}(\textbf{new}(), 1) \textbf{ in}$$
$$\text{let } \ell, n \ = \ \textbf{acquire}(\ell) \textbf{ in}$$
$$\text{let } \ell \ = \ \textbf{release}(\ell, n + 1) \textbf{ in}$$
$$\text{let } m \ = \ \textbf{wait}(\ell)$$

If the values we store in the cell are of unrestricted type (duplicable and droppable), we can implement references with **ref**, **get**, and **set** as follows:

$$\textbf{ref} : \tau \to \textbf{Lock}\langle\tau_o^1\rangle$$
$$\textbf{ref}(v) \triangleq \textbf{release}(\textbf{new}(), v)$$

$$\textbf{get} : \textbf{Lock}\langle\tau_o^a\rangle \to \textbf{Lock}\langle\tau_o^a\rangle \times \tau \quad \text{where } \tau \text{ unr}$$
$$\textbf{get}(\ell) \triangleq \textbf{let } \ell, v = \textbf{acquire}(\ell) \textbf{ in } (\textbf{release}(\ell, v), v)$$

$$\textbf{set} : \textbf{Lock}\langle\tau_o^a\rangle \times \tau \to \textbf{Lock}\langle\tau_o^a\rangle \quad \text{where } \tau \text{ unr}$$
$$\textbf{set}(\ell, v) \triangleq \textbf{let } \ell, v' = \textbf{acquire}(\ell) \textbf{ in release}(\ell, v)$$

Similarly, we can atomically exchange the value as follows or apply a function to the value as follows:

$$\textbf{xchng} : \textbf{Lock}\langle\tau_o^a\rangle \times \tau \to \textbf{Lock}\langle\tau_o^a\rangle \times \tau$$
$$\textbf{xchng}(\ell, v) \triangleq \textbf{let } \ell, v' = \textbf{acquire}(\ell) \textbf{ in } (\textbf{release}(\ell, v), v')$$

$$\textbf{modify} : \textbf{Lock}\langle\tau_o^a\rangle \times (\tau \multimap \tau) \to \textbf{Lock}\langle\tau_o^a\rangle$$
$$\textbf{modify}(\ell, f) \triangleq \textbf{let } \ell, v = \textbf{acquire}(\ell) \textbf{ in release}(\ell, f\ v)$$

These operations work even for linear values, since neither v nor v' are duplicated or dropped.

Fork-join / futures / promises

In the generalised fork-join model with futures / promises, the parent thread can spawn a child thread to do some work, and later synchronize with the child to obtain the result. Our lock operations directly support this:

- The parent thread creates a new lock using $\ell = \textbf{new}()$.

- The parent forks off the child thread, sharing an opened client reference to ℓ with the child.

- The parent thread continues doing other work, and eventually calls **wait**(ℓ) on the lock.

- When the child thread is done with the work, it calls **release**(ℓ, v) with the result v.

This is illustrated in the following program:

$$\textbf{let } \ell = \textbf{fork}(\textbf{new}(), \lambda\ell. \cdots \textbf{drop}(\textbf{release}(\ell, v)) \cdots) \textbf{ in}$$
$$\cdots$$
$$\textbf{let } v = \textbf{wait}(\ell)$$

Note how the type system ensures deadlock and leak freedom:

- Initially, **new** : **Lock**$\langle \tau_1^1 \rangle$, *i.e.,* the lock is an open owner reference.

- When we fork, we split the lock up into **Lock**$\langle \tau_0^1 \rangle$ and **Lock**$\langle \tau_1^0 \rangle$.

- The closed and owning reference of type **Lock**$\langle \tau_0^1 \rangle$ goes to the parent, who eventually waits for the result.

- The open and client reference of type **Lock**$\langle \tau_1^0 \rangle$ goes to the child, who must put a value in it in order to drop it.

Of course, the client is free to pass around its reference to the lock, which acts as a future/promise, so that somebody else can fulfill the obligation to release the lock by putting a value in it.

Concurrently shared data

The parent thread can spawn multiple child threads and create a new lock for each, as in the fork-join pattern. However, the parent can also create one lock, put an initial data structure v in it, and share lock references with several children, who may each acquire and mutate the lock's contents repeatedly:

$$
\begin{aligned}
&\textbf{let } \ell \ = \ \textbf{release}(\textbf{new}(), v) \textbf{ in} \\
&\textbf{let } \ell \ = \ \textbf{fork}(\ell, \lambda\ell. \ \cdots) \textbf{ in} \\
&\textbf{let } \ell \ = \ \textbf{fork}(\ell, \lambda\ell. \ \cdots) \textbf{ in} \\
&\textbf{let } \ell \ = \ \textbf{fork}(\ell, \lambda\ell. \ \cdots) \textbf{ in} \\
&\cdots \\
&\textbf{let } v' \ = \ \textbf{wait}(\ell)
\end{aligned}
$$

Children are of course free to fork off children of their own, all sharing access to the same lock ℓ.

Bank example

Consider three bank accounts whose balances are stored in locks ℓ_1, ℓ_2, ℓ_3. The main thread acts as the bank, spawns three clients, and gives them access to their bank account so that they can deposit and withdraw money from it:

$$
\begin{aligned}
&\textbf{let } \ell_1 \ = \ \textbf{fork}(\textbf{release}(\textbf{new}(), 0), \lambda\ell_1. \ \cdots \text{client } 1 \cdots) \textbf{ in} \\
&\textbf{let } \ell_2 \ = \ \textbf{fork}(\textbf{release}(\textbf{new}(), 0), \lambda\ell_2. \ \cdots \text{client } 2 \cdots) \textbf{ in} \\
&\textbf{let } \ell_3 \ = \ \textbf{fork}(\textbf{release}(\textbf{new}(), 0), \lambda\ell_3. \ \cdots \text{client } 3 \cdots) \textbf{ in} \\
&\cdots \\
&\textbf{let } \ell_1, \ell_2 \ = \ \text{transaction}(\ell_1, \ell_2, 50) \textbf{ in} \ \cdots
\end{aligned}
$$

The bank does a transaction between ℓ_1 and ℓ_2:

transaction : $\mathbf{Lock}\langle\mathbf{int}_o^a\rangle \times \mathbf{Lock}\langle\mathbf{int}_o^a\rangle \times \mathbf{int} \multimap \mathbf{Lock}\langle\mathbf{int}_o^a\rangle \times \mathbf{Lock}\langle\mathbf{int}_o^a\rangle$

transaction$(\ell_1, \ell_2, \mathsf{amount}) \triangleq$

    let $\ell_1, \mathsf{balance}_1 = \mathbf{acquire}(\ell_1)$ **in**

    let $\ell_2, \mathsf{balance}_2 = \mathbf{acquire}(\ell_2)$ **in**

    if $\mathsf{balance}_1 \geqslant \mathsf{amount}$ **then**

      $(\mathbf{release}(\ell_1, \mathsf{balance}_1 - \mathsf{amount}), \mathbf{release}(\ell_2, \mathsf{balance}_2 + \mathsf{amount}))$

    else

      $(\mathbf{release}(\ell_1, \mathsf{balance}_1), \mathbf{release}(\ell_2, \mathsf{balance}_2))$

Note that we did not have to keep track of any lock orders, or had to do any other analysis to show that this does not deadlock, regardless of what the rest of the program does. In Section 2.6 we introduce lock groups, which allow us to extend this example to multiple bank threads sharing multiple locks, still ensuring deadlock and memory leak freedom.

Shared mutable recursive data structures

We can define a recursive type tree where each node is protected by a lock and stores a value of type τ:

$$\mathsf{tree} \triangleq \mathbf{Lock}\langle 1 + \mathsf{tree} \times \tau \times \mathsf{tree}_o^1\rangle$$

These trees own their children. In order to operate over such trees conurrently, we define the type tree' of client references to trees:

$$\mathsf{tree}' \triangleq \mathbf{Lock}\langle 1 + \mathsf{tree} \times \tau \times \mathsf{tree}_o^o\rangle$$

The main thread can now allocate a tree, and share multiple client references of type tree' with child threads. Using a client reference we can not only modify the root, but we can also traverse the tree. For instance, to try and obtain a client reference to

the left child (if any), we acquire the lock, create a client reference to the left child (using **fork**), and release the lock:

$$\text{left} : \text{tree}' \multimap 1 + \text{tree}'$$
$$\text{left}(\ell) \triangleq$$

> **let** $\ell, t = $ **acquire**(ℓ) **in**
> **match** t **with**
> $\quad \text{in}_{\mathbf{L}}() \Rightarrow \textbf{release}(\ell, \text{in}_{\mathbf{L}}()); \text{in}_{\mathbf{L}}()$
> $\quad \text{in}_{\mathbf{R}}(\ell_1, x, \ell_2) \Rightarrow \text{in}_{\mathbf{R}}(\textbf{fork}(\ell_1, \lambda\ell_1.\ \textbf{release}(\ell, \text{in}_{\mathbf{R}}(\ell_1, x, \ell_2))))$
> **end**

Note that **fork** operates as an administrative device here; when we acquire the lock ℓ we obtain owning references ℓ_1, ℓ_2 to the children, and fork allows us to obtain a client reference for ℓ_1 while putting the owning references back in the lock ℓ. One would not actually fork a thread in a real implementation.

Because we immediately release the lock after obtaining a child reference, we can have multiple threads operate on different parts on the tree concurrently, while guaranteeing leak and deadlock freedom.

Client-server

Our language also comes equipped with linear channels for message-passing concurrency that can be session-typed, thanks to our encoding detailed in Section 2.3.1. Using locks, we can share a channel endpoint among multiple participants, which allows us to implement a client-server pattern, as is possible in the deadlock-free fragment of manifest sharing (Balzer and Pfenning, 2017).

> **let** $c = (\cdots \text{create new server channel} \cdots)$ **in**
> **let** $\ell = \textbf{release}(\textbf{new}(), c)$ **in**
> **let** $\ell = \textbf{fork}(\ell, \lambda\ell.\ \cdots)$ **in**
> **let** $\ell = \textbf{fork}(\ell, \lambda\ell.\ \cdots)$ **in**
> $\cdots$
> **let** $c = \textbf{wait}(\ell)$

Each client can temporarily take the lock, which allows it to interact with the server. As in (Balzer et al., 2019), typing ensures that a lock must be released to the same protocol state at which it was previously acquired, ensuring type safety.

Locks over channels

The preceding example involves putting channels in locks, but we can also send locks over channels. For instance, one can send an open lock acting like a future/promise

to another thread, so that the other thread gets the obligation to fulfill the promise by storing a value in the lock.

Encoding session-typed channels

In Section 2.3.1 we show that we can implement session-typed channels using our locks.

2.2.4 *Sharing Multiple Locks with Lock Orders*

The simple system illustrated above is restricted to sharing only one lock at each fork. We lift this restriction in Section 2.6 by introducing *lock groups*. Lock groups consist of multiple locks, and one is allowed to share an entire lock group at each fork. In turn, we must introduce another mechanism to ensure leak and deadlock freedom within a lock group. We do this by imposing a lock order on the locks of a lock group, and requiring that the locks are acquired in increasing order. A similar condition takes care that there is no deadlock between several **wait**s or between **wait** and **acquire**. In Section 2.6.1 we provide examples of the use of lock orders. In particular, we can handle a version of Dijkstra's dining philosophers problem with a *dynamic* number of participants dependent on a run-time variable n.

Importantly, deadlock freedom between lock groups is taken care of by the sharing topology, so one is always free to acquire locks from different lock groups, and do transactions between different lock groups in that manner. This makes lock groups more *compositional* than standard global lock orders that require a global order on the entire system whenever multiple locks are acquired.

2.3 THE λ_{LOCK} LANGUAGE

We give a formal description of λ_{lock}'s syntax, type system, and operational semantics. The base of λ_{lock} is a linear λ-calculus, extended with unrestricted types (whose values can be freely duplicated, dropped, and deallocated) and recursive types:

$$\tau \in \textbf{Type} \triangleq \textbf{0} \mid \textbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid \textbf{Lock}\langle \tau \, {}^a_b \rangle \mid \mu x.\tau \mid x \qquad \text{⚙}$$

We distinguish linear functions $\tau_1 \multimap \tau_2$ from unrestricted functions $\tau_1 \to \tau_2$. Unrestricted functions can be freely duplicated and discarded, and hence can only capture unrestricted variables. Linear functions, on the other hand, must be treated linearly, and hence can close over both linear and unrestricted variables. Rather than distinguishing sums and products into linear and unrestricted, we consider sums and products to be unrestricted if their components are. Similarly, we consider recursive types to be unrestricted if their coinductive unfoldings are (see Section 2.7). The empty type $\textbf{0}$ and unit type $\textbf{1}$ are always unrestricted. The lock type $\textbf{Lock}\langle \tau \, {}^a_b \rangle$ is always linear, regardless of whether τ is.

$$\frac{\Gamma \text{ unr}}{\Gamma \vdash \mathbf{new}() : \mathbf{Lock}\langle \tau_0^1 \rangle} \qquad \frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^0 \rangle}{\Gamma \vdash \mathbf{drop}(e) : \mathbf{1}} \qquad \frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^1 \rangle}{\Gamma \vdash \mathbf{wait}(e) : \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau_{b_1+b_2}^{a_1+a_2} \rangle \qquad \Gamma_2 \vdash e_2 : \mathbf{Lock}\langle \tau_{b_2}^{a_2} \rangle \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}(e_1, e_2) : \mathbf{Lock}\langle \tau_{b_1}^{a_1} \rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau_0^a \rangle}{\Gamma \vdash \mathbf{acquire}(e) : \mathbf{Lock}\langle \tau_1^a \rangle \times \tau} \qquad \frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau_1^a \rangle \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{release}(e_1, e_2) : \mathbf{Lock}\langle \tau_0^a \rangle}$$

Figure 12: λ_{lock}'s lock typing rules.

Our language λ_{lock} has the following syntax:

$$e \in \text{Expr} ::= x \mid () \mid (e, e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{let}\ (x_1, x_2)\ =\ e\ \mathbf{in}\ e \mid$$
$$\mathbf{match}\ e\ \mathbf{with}\ \bot\ \mathbf{end} \mid \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} \mid$$
$$\mathbf{new}() \mid \mathbf{fork}(e, e) \mid \mathbf{acquire}(e) \mid \mathbf{release}(e, e) \mid \mathbf{drop}(e) \mid \mathbf{wait}(e)$$

The typing rules for the lock operations can be found in Figure 12, and the typing rules for the base language can be found in Figure 13. We use the judgments Γ unr and $\Gamma \equiv \Gamma_1 \cdot \Gamma_2$ to handle linear and unrestricted types: Γ unr means that all types in Γ are unrestricted, and $\Gamma \equiv \Gamma_1 \cdot \Gamma_2$ splits up Γ into Γ_1 and Γ_2 disjointly for variables of linear type, while allowing variables of unrestricted type to be shared by both Γ_1 and Γ_2. We do not include a constructor for recursive functions, because recursive functions can already be encoded in terms of recursive types, using the Y-combinator.[1]

The rules for the operational semantics can be found in Figure 14. We use a small step operational semantics built up in two layers. The first layer defines values, evaluation contexts, and reductions for pure expressions. The values are standard for λ-calculus, except for $\langle k \rangle$, which indicates a reference/pointer to a lock identified by the number k.

The second layer operates on a *configuration*, which is a collection of threads and locks, each identified with a natural number. A thread $\mathsf{Thread}(e)$ comprises the expression e that it executes, and a lock $\mathsf{Lock}(\text{refcnt}, \mathbf{None} \mid \mathbf{Some}(v))$ comprises a reference count refcnt (*i.e.*, the number of client references) and either $\mathbf{None}$, indicating that the lock has been acquired and currently contains no value, or $\mathbf{Some}(v)$, indicating that the lock is currently closed and is holding the value v.

The stepping rules for the configuration are as follows, as labeled in Figure 14.

PURE Perform a pure reduction in an evaluation context.

---

1 Of course, for efficiency of an implementation one wants direct support for recursion.

$$\frac{\Gamma \text{ unr}}{\Gamma, x{:}\tau \vdash x : \tau} \qquad\qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \text{ unr} \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \text{ unr}}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma \vdash \mathbf{let}\ x_1, x_2\ =\ e_1\ \mathbf{in}\ e_2 : \tau_3}$$

$$\frac{\Gamma \text{ unr} \qquad \Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \bot\ \mathbf{end} : \tau} \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{in_L}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{in_R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau' \qquad \Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau'}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} : \tau'}$$

Figure 13: λ_lock's base linear λ-calculus with sums & products and linear & unrestricted functions.

$$v \in \text{Val} ::= () \mid (v, v) \mid \textbf{in}_{\textbf{L}}(v) \mid \textbf{in}_{\textbf{R}}(v) \mid \lambda x.\, e \mid \langle k \rangle \qquad \text{✿}$$

$$K \in \text{Ctx} ::= \square \mid (K, e) \mid (v, K) \mid \textbf{in}_{\textbf{L}}(K) \mid \textbf{in}_{\textbf{R}}(K) \mid K\, e \mid v\, K \mid \textbf{let } x_1, x_2 \; = \; K \textbf{ in } e \qquad \text{✿}$$
$$\mid \textbf{match } K \textbf{ with } \bot \textbf{ end} \mid \textbf{match } K \textbf{ with in}_{\textbf{L}}(x_1) \Rightarrow e_1; \; \textbf{in}_{\textbf{R}}(x_2) \Rightarrow e_2 \textbf{ end}$$
$$\mid \textbf{new}(K) \mid \textbf{fork}(K, e) \mid \textbf{fork}(v, K) \mid \textbf{acquire}(K) \mid \textbf{release}(K, e)$$
$$\mid \textbf{release}(v, K) \mid \textbf{drop}(K) \mid \textbf{wait}(K)$$

$$\textbf{match in}_{\textbf{L}}(v) \textbf{ with in}_{\textbf{L}}(x_1) \Rightarrow e_1 \mid \textbf{in}_{\textbf{R}}(x_2) \Rightarrow e_2 \textbf{ end} \leadsto_{\text{pure}} e_1[v/x_1] \qquad \text{✿}$$
$$\textbf{match in}_{\textbf{R}}(v) \textbf{ with in}_{\textbf{L}}(x_1) \Rightarrow e_1 \mid \textbf{in}_{\textbf{R}}(x_2) \Rightarrow e_2 \textbf{ end} \leadsto_{\text{pure}} e_2[v/x_2]$$
$$\textbf{let } x_1, x_2 \; = \; (v_1, v_2) \textbf{ in } e \leadsto_{\text{pure}} e[v_1/x_1][v_2/x_2]$$
$$(\lambda x.\, e)\, v \leadsto_{\text{pure}} e[v/x]$$

$$\rho \in \text{Cfg} \triangleq N \xrightarrow{\text{fin}} \text{Thread}(e) \mid \text{Lock}(\text{refcnt}, \textbf{None} \mid \textbf{Some}(v)) \qquad \text{✿✿}$$

$$e_1 \leadsto_{\text{pure}} e_2 \implies \left\{ n \mapsto \text{Thread}(K[\, e_1\,]) \right\} \overset{n}{\leadsto} \left\{ n \mapsto \text{Thread}(K[\, e_2\,]) \right\} \qquad \text{(pure)}$$

$$\left\{ n \mapsto \text{Thread}(K[\, \textbf{new}()\,]) \right\} \overset{n}{\leadsto} \left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \langle k \rangle\,]) \\ k \mapsto \text{Lock}(0, \textbf{None}) \end{matrix} \right\} \qquad \text{(new)}$$

$$\left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \textbf{fork}(\langle k \rangle, v)\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, x) \end{matrix} \right\} \overset{k}{\leadsto} \left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \langle k \rangle\,]) \\ m \mapsto \text{Thread}(v\, \langle k \rangle) \\ k \mapsto \text{Lock}(1 + \text{refcnt}, x) \end{matrix} \right\} \qquad \text{(fork)}$$

$$\left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \textbf{acquire}(\langle k \rangle)\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, \textbf{Some}(v)) \end{matrix} \right\} \overset{k}{\leadsto} \left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, (\langle k \rangle, v)\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, \textbf{None}) \end{matrix} \right\} \qquad \text{(acquire)}$$

$$\left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \textbf{release}(\langle k \rangle, v)\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, \textbf{None}) \end{matrix} \right\} \overset{k}{\leadsto} \left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \langle k \rangle\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, \textbf{Some}(v)) \end{matrix} \right\} \qquad \text{(release)}$$

$$\left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \textbf{drop}(\langle k \rangle)\,]) \\ k \mapsto \text{Lock}(1 + \text{refcnt}, x) \end{matrix} \right\} \overset{k}{\leadsto} \left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, ()\,]) \\ k \mapsto \text{Lock}(\text{refcnt}, x) \end{matrix} \right\} \qquad \text{(drop)}$$

$$\left\{ \begin{matrix} n \mapsto \text{Thread}(K[\, \textbf{wait}(\langle k \rangle)\,]) \\ k \mapsto \text{Lock}(0, \textbf{Some}(v)) \end{matrix} \right\} \overset{k}{\leadsto} \left\{ n \mapsto \text{Thread}(K[\, v\,]) \right\} \qquad \text{(wait)}$$

$$\left\{ n \mapsto \text{Thread}(()) \right\} \overset{n}{\leadsto} \left\{ \right\} \qquad \text{(exit)}$$

$$\rho_1 \overset{i}{\leadsto} \rho_2 \implies \rho_1 \uplus \rho' \overset{i}{\leadsto} \rho_2 \uplus \rho' \qquad \text{(frame)}$$

Figure 14: λ_{lock}'s operational semantics.

NEW Allocate a new lock at a fresh position k, and return a reference ⟨k⟩ to the thread.

FORK Fork off a new thread, while duplicating the reference to lock k, passing ⟨k⟩ back to the main thread, as well as to the new child thread.

ACQUIRE If the lock currently contains **Some**(v), then the acquire can proceed, and returns the value to the thread and puts **None** in the lock.

RELEASE Does the opposite: replaces **None** in the lock with **Some**(v), where v is the value provided to the release operation.

DROP Deletes a reference to the lock, decrementing its reference count.

WAIT When the reference count is 0 and there is a **Some**(v) in the lock, the operation can proceed and removes the lock from the configuration, while giving the value to the thread.

EXIT When a thread has terminated with a unit value, we remove the thread from the configuration.

FRAME Closes the set of preceding rules under disjoint union with a remaining configuration. This allows the preceding rules to take place within a large configuration.

2.3.1 *Encoding Session-Typed Channels*

One can implement session-typed channels using our locks. Consider basic session types (Honda, 1993; Wadler, 2012; Lindley and Morris, 2015):

$$s \in \mathsf{Session} ::= \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \,\&\, s \mid s \oplus s \mid \mathsf{End}_! \mid \mathsf{End}_?$$

We can implement the usual channel operations as follows:

$$
\begin{aligned}
\mathbf{fork}_C(f) &\triangleq \mathbf{fork}(\mathbf{new}(), f) \\
\mathbf{send}_C(c, v) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, (c', v)))) \\
\mathbf{receive}_C(c) &\triangleq \mathbf{wait}(c) \\
\mathbf{tell}_L(c) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, \mathbf{in}_L(c')))) \\
\mathbf{tell}_R(c) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, \mathbf{in}_R(c')))) \\
\mathbf{ask}(c) &\triangleq \mathbf{wait}(c) \\
\mathbf{close}_C(c) &\triangleq \mathbf{drop}(\mathbf{release}(c, ())) \\
\mathbf{wait}_C(c) &\triangleq \mathbf{wait}(c)
\end{aligned}
$$

Of course, implementing channels this way is inefficient, because a tiny thread is forked every time we send a message. Thus, it is still worth having native channels,

or a compiler that supports a version of fork that does not actually spawn a new thread, but runs the body immediately (though care must be taken not to introduce deadlocks). To type these operations, we use the following definition of session types in terms of locks, where $\overline{s}$ is the dual of s:

$$!\tau.s \triangleq \mathbf{Lock}\langle \overline{s} \times \tau_1^0 \rangle$$
$$?\tau.s \triangleq \mathbf{Lock}\langle s \times \tau_0^1 \rangle$$
$$s_1 \mathbin{\&} s_2 \triangleq \mathbf{Lock}\langle \overline{s_1} + \overline{s_2}\,_1^0 \rangle$$
$$s_1 \oplus s_2 \triangleq \mathbf{Lock}\langle s_1 + s_2\,_0^1 \rangle$$
$$\mathsf{End}_! \triangleq \mathbf{Lock}\langle \mathbf{1}\,_1^0 \rangle$$
$$\mathsf{End}_? \triangleq \mathbf{Lock}\langle \mathbf{1}\,_0^1 \rangle$$

This encoding resembles the encodings of (Kobayashi, 2002b; Dardha et al., 2012, 2017). After encoding session types this way, we can type check the channel operations with the standard session typing rules:

$$\mathbf{fork}_C : (s \multimap \mathbf{1}) \multimap \overline{s}$$
$$\mathbf{close}_C : \mathsf{End}_! \multimap \mathbf{1}$$
$$\mathbf{wait}_C : \mathsf{End}_? \multimap \mathbf{1}$$
$$\mathbf{send}_C : !\tau.s \times \tau \multimap s$$
$$\mathbf{receive}_C : ?\tau.s \multimap s \times \tau$$
$$\mathbf{tell}_L : s_1 \mathbin{\&} s_2 \multimap s_1$$
$$\mathbf{tell}_R : s_1 \mathbin{\&} s_2 \multimap s_2$$
$$\mathbf{ask} : s_1 \oplus s_2 \multimap s_1 + s_2$$

Because we can encode these session-typed channels in our deadlock and memory leak free language, this automatically shows that these session-typed channels are deadlock and memory leak free. Note that the encoding of session types relies in an essential way on higher-order locks.

2.4 THE DEADLOCK AND LEAK FREEDOM THEOREMS

Our goal was to make λ_{lock} deadlock and memory leak free. We will now make this more precise. Firstly, let us look at how the usual notion of type safety can be adapted to a language with blocking constructs. Type safety for a single threaded language says that if we start with a well-typed program, then the execution of the program does not get stuck until we terminate with a value. If we have multiple threads, we could say that this has to hold *for every thread*, but if we have blocking constructs this is clearly not true: a thread can temporarily get stuck while blocking. We therefore modify the notion of type safety to say that each thread can always make progress,

except if the thread is legitimately blocked, *i.e.,* blocked on an operation that is supposed to block, like **acquire**.

This, of course, is not a strong enough property for our purposes. To rule out deadlocks, we want to say that even if *some* threads are blocked, there is always *at least one* thread that can make progress. Furthermore, we wish to say that if all threads terminate, then all memory has been deallocated. Because of the way we have set up our operational semantics, we can formulate this simply as saying: if the configuration cannot step, then it must be empty.

Let us consider the formal statement of global progress:

Theorem 2.4.1 (Global progress ⚙).

   If $\emptyset \vdash e : \mathbf{1}$, *and* $\left\{ o \mapsto \mathsf{Thread}(e) \right\} \leadsto^* \rho$, *then either* $\rho = \left\{ \right\}$ *or* $\exists \rho'. \rho \leadsto \rho'$.

Global progress rules out whole-program deadlocks, and it ensures that all locks have been deallocated when the program terminates. However, it does not guarantee anything as long as there is still a single thread that can step. Thus it only guarantees a weak form of deadlock freedom, and it only guarantees memory leak freedom when the program terminates, not during execution.

To formulate stronger forms of deadlock and leak freedom, we take an approach similar to the approaches previously taken for session types (Jacobs et al., 2022b). Namely, we define the relation $i \mathrel{\mathsf{waiting}_\rho} j$, which says that $i \in \mathrm{dom}(\rho)$ is waiting for $j \in \mathrm{dom}(\rho)$. Intuitively, in the graph of connections between objects in the configuration (*i.e.,* between threads and locks, and between locks and locks), we give each such connection a waiting direction, so that either $i \mathrel{\mathsf{waiting}_\rho} j$, or $j \mathrel{\mathsf{waiting}_\rho} i$. We define this relation such that if i is a thread, and currently about to execute a lock operation, then $i \mathrel{\mathsf{waiting}_\rho} j$. Furthermore, in all other cases, we say that $j \mathrel{\mathsf{waiting}_\rho} i$, if there is some reference from i to j or from j to i.

Consider our operational semantics stepping rule $\rho \overset{i}{\leadsto} \rho'$: this step relation is annotated with a number i, indicating which object in the configuration we consider responsible for the step. The waiting relation sets up a blame game with respect to this step relation: whenever we ask some object i why the configuration isn't making progress, i should either respond that it can make the configuration step (*i.e.,* $\exists \rho', \rho \overset{i}{\leadsto} \rho'$), or i should blame somebody else, by showing $\exists j, i \mathrel{\mathsf{waiting}_\rho} j$.

We can then continue to ask j why the configuration isn't making progress, and so on. Since we maintain the invariant that the graph of connections is acyclic, it is not possible that the blame game loops back to the original i in a cycle, since then we'd either have a cycle in the reference structure. Furthermore, the blame game cannot revisit i via the same edge that was used to leave i either, since then we'd have $i \mathrel{\mathsf{waiting}_\rho} j$ and $j \mathrel{\mathsf{waiting}_\rho} i$ for some j, which is impossible due to the way we've defined $\mathsf{waiting}_\rho$. Therefore we conclude that the blame game must eventually terminate at some $j \in \mathrm{dom}(\rho)$ who shows that the configuration can step.

Importantly, this gives us a stronger theorem, namely that if we start at any $i \in \mathrm{dom}(\rho)$, there is some j *transitively connected to i via waiting dependencies*, such that

j can make the configuration step. This will rule out that a subset of the configuration has been leaked or deadlocked, because in that case there would be no such transitive path to a thread that can step.

In contrast to Jacobs et al. (2022b), we define these notions more generically, so that we only need to prove *one* language specific theorem, from which all the other properties that we wish to establish follow generically, without further dependence on the particular language under consideration.

Let us now look at this in more formal detail. A language specific piece of information we need is the relation e blocked k, which says that expression e is blocked on the object identified by k. Note that unlike the waiting relation, this relation does not depend on the configuration; whether an expression e is blocked can be determined from the expression itself:

Definition 2.4.2. ✿ We have e blocked k if e is of the form $K[\, e_o \,]$ for some evaluation context K, and e_o is one of **fork**($\langle k \rangle$, v), **acquire**($\langle k \rangle$), **release**($\langle k \rangle$, v), **drop**($\langle k \rangle$), **wait**($\langle k \rangle$), for some value v.

Note that we formally include all the lock operations in the blocked relation, even the ones that are conventionally not thought of as blocking. The reason we do this is because we consider the locks to be responsible for the step operations involving the lock, and not the thread, as can be seen from the annotations i on the step relation $\rho \overset{i}{\leadsto} \rho'$ in the operational semantics (Figure 14). This streamlines the formal statements because they become more uniform.

Secondly, we need to be able to determine all the outgoing references for an object in the configuration:

Definition 2.4.3. ✿ We have the function $\text{refs}_\rho(i) \subseteq \text{dom}(\rho)$, which gives the set of all references $\langle k \rangle$ stored inside $\rho(i)$. We omit the formal definition here, as this can be defined using a straightforward recursion on expressions and values.

This allows us to define the waiting relation:

Definition 2.4.4. ✿ We have i waiting$_\rho$ j if either:

1. $\rho(i) = \text{Thread}(e)$ and e blocked j.

2. $i \in \text{refs}_\rho(j)$ and not $\rho(j) = \text{Thread}(e)$ with e blocked i.

That is, threads are waiting for the objects they are blocked on, and if an object has an incoming reference and this reference is not from a thread blocked on that object, then the object is considered to be waiting for the source of the incoming reference. Specifically, if a thread has a reference to a lock, and the thread is not currently about to execute an operation with this lock, then the lock is said to be waiting for the thread. Similarly, if a lock holds a reference to a lock, then the second lock is considered to be waiting for the first.

Using the waiting relation notion, we can formally define what a partial deadlock/leak is. Intuitively, a partial deadlock is a subset of the objects in the

configuration, none of which can step, such that if an object in the deadlock is waiting, then it must be waiting for another object in the deadlock.

Definition 2.4.5 (Partial deadlock/leak ✿). Given a configuration ρ, a non-empty subset $S \subseteq \text{dom}(\rho)$ is in a partial deadlock/leak if these two conditions hold:

1. No $i \in S$ can step, *i.e.*, for all $i \in S$, $\neg\exists\rho'. \rho \overset{i}{\rightsquigarrow} \rho'$

2. If $i \in S$ and $i \text{ waiting}_\rho j$ then $j \in S$

This notion also incorporates memory leaks: if there is some lock that is not referenced by a thread or other lock, then the singleton set of that lock is a partial deadlock/leak. Furthermore, if we have a set of locks that all reference only each other circularly, then this is considered to be a partial deadlock/leak. Similarly, a single thread that is not synchronizing on a lock, is considered to be in a singleton deadlock if it cannot step.

Definition 2.4.6 (Partial deadlock/leak freedom ✿). A configuration ρ is deadlock/leak free if no $S \subseteq \text{dom}(\rho)$ is in a partial deadlock/leak.

We can reformulate this to look more like the standard notion of memory leak freedom, namely reachability:

Definition 2.4.7 (Reachability ✿). We inductively define the threads and locks that are *reachable* in ρ: $j_0 \in \mathbb{N}$ is reachable in ρ if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geqslant 0$) such that $j_0 \text{ waiting}_\rho j_1$, and $j_1 \text{ waiting}_\rho j_2$, ..., and $j_{k-1} \text{ waiting}_\rho j_k$, and finally j_k can step in ρ, *i.e.*, $\exists\rho'. \rho \overset{j_k}{\rightsquigarrow} \rho'$.

Intuitively, an element of the configuration is reachable if it can step, or if it has a transitive waiting dependency on some other element that can step. This notion is stronger than the usual notion of reachability, which considers objects to be reachable even if they are only reachable from threads that are blocked.

Definition 2.4.8 (Full reachability ✿). A configuration ρ is *fully reachable* if all $i \in \text{dom}(\rho)$ are reachable in ρ.

As in (Jacobs et al., 2022b), our strengthened formulations of deadlock freedom and full reachability are equivalent for λ_{lock}:

Theorem 2.4.9. ✿ *A configuration ρ is deadlock/leak free if and only if it is fully reachable.*

In contrast to (Jacobs et al., 2022b), we have carefully set up our definitions so that this theorem holds fully generically, *i.e.*, independent of any language specific properties.

These notions also imply global progress and type safety:

Definition 2.4.10. ✿

A configuration ρ satisfies global progress if $\rho = \emptyset$ or $\exists\rho', i. \rho \overset{i}{\rightsquigarrow} \rho'$.

Definition 2.4.11. ⚙

A configuration ρ is safe if for all $i \in \text{dom}(\rho)$, $\exists \rho', i. \rho \overset{i}{\leadsto} \rho'$, or $\exists j. i \text{ waiting}_\rho j$.

That is, global progress holds if there is any element of the configuration that can step, and safety holds if all elements of the configuration can either step, or are legally blocked (*i.e.,* waiting for someone else).

Theorem 2.4.12. ⚙ ⚙

If a configuration ρ is fully reachable, then ρ has the progress and safety properties.

We thus only need to prove one language specific theorem, namely that all configurations that arise from well-typed programs are fully reachable:

Theorem 2.4.13. ⚙

If $\emptyset \vdash e : \mathbf{1}$ and $\left\{ o \mapsto \text{Thread}(e) \right\} \leadsto^* \rho'$, then ρ' is fully reachable.

Once we have this theorem, the other theorems follow.

In the next section (Section 2.5) we give a high-level overview of how the theorem is proved.

2.5 AN INTUITIVE DESCRIPTION OF THE PROOFS

In this section we give a high-level overview of the proof of Theorem 2.4.13. We keep the discussion high-level and intuitive because the full details are in the mechanized proofs (Section 2.7).

Recall that Theorem 2.4.13 says that if we start with a well-typed program, then every object in the configuration always remains reachable (Theorem 2.4.7). In order to show this, we will structure the proof in the style of progress and preservation: we first define an invariant on the configuration, showing that the invariant is preserved by the operational semantics (analogous to preservation), and then show that configurations that satisfy the invariant are fully reachable (analogous to progress). Thus, our first task is to come up with a suitable invariant.

As we have seen in Section 2.2, our invariant must ensure that the sharing topology in the configuration is acyclic. That is, if one considers the graph where the threads and locks are vertices, and there is an (undirected) edge between two vertices if there is a reference between them (in either direction), then this graph shall remain acyclic.

Another aspect of our invariant is well-typedness: we must ensure that the expressions of every thread, and the values in every lock, are well-typed. Furthermore, if there are lock references $\langle k \rangle$ in expressions or values, then the type assigned to these must be consistent with the type of values actually stored in the lock.

However, the type of lock references is, in general, $\langle k \rangle : \mathbf{Lock}\langle \tau \, {}^a_b \rangle$. The invariant must also account for the consistency of the a and b of the different references to the same lock. We require the following conditions for a lock $\left\{ k \mapsto \text{Lock}(\text{refcnt}, v) \right\}$ in the configuration:

- Out of all the references $\langle k \rangle$ appearing in the configuration, precisely one has $a = 1$.

- Out of all the references $\langle k \rangle$ appearing in the configuration, at most one has $b = 1$. Furthermore, if $v = \textbf{Some}(v')$, we must have $b = 0$ for all references, and if $v = \textbf{None}$, we must have precisely one reference with $b = 1$.

- The number of references with $a = 0$ must be consistent with refcnt.

We capture the acyclicity condition, the well-typedness condition, and the lock conditions in a predicate $\mathsf{inv}(\rho)$ on configurations, which states that the configuration ρ satisfies these conditions. Our invariant is that this predicate holds throughout execution. Formally, we have to show:

Theorem 2.5.1 (Preservation ✿). *If* $\rho \overset{i}{\leadsto} \rho'$ *then* $\mathsf{inv}(\rho) \implies \mathsf{inv}(\rho')$

The proof of this theorem involves careful mathematical reasoning about the sharing topology: we must show that each lock operation, although it may modify the structure of the graph, ensures that if the graph of ρ was acyclic, then the graph of ρ' is also acyclic, provided the program we are executing is well-typed in the linear type system.

Secondly, we must ensure that all operations leave all the expressions and values in the configuration well-typed, and maintain the correctness conditions for the references to each lock.

Having done this, it remains to show that if a configuration satisfies our invariant, then every object in the configuration is reachable:

Theorem 2.5.2 (Reachability ✿). *If* $\mathsf{inv}(\rho)$, *then* ρ *is fully reachable.*

Recall Theorem 2.4.7 of reachability: an object j_0 in the configuration is reachable, if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geqslant 0$) such that j_0 waiting$_\rho$ j_1, and j_1 waiting$_\rho$ j_2, ..., and j_{k-1} waiting$_\rho$ j_k, and finally j_k can step in ρ, *i.e.*, $\exists \rho'. \rho \overset{j_k}{\leadsto} \rho'$. Thus, proving Theorem 2.5.2 amounts to constructing such a sequence and showing that the final element in the sequence can step. To construct this sequence, we must rely on the acyclicity of the sharing topology in an essential way.

We do this by formulating our strategy for constructing such a sequence with respect to that graph: we start at the vertex j_0, check whether j_0 can itself step, and if not, show that there must exist some j_1 such that j_0 waiting$_\rho$ j_1. We then repeat this process iteratively.

There is a danger that this process does not terminate (*i.e.*, goes in a cycle, since the configuration is finite), but by being careful we can avoid this danger:

1. We make sure, that if we step from j_i to j_{i+1}, that there is an edge between j_i and j_{i+1}.

2. We make sure that if we just stepped from j_i to j_{i+1}, we will not immediately step back.

These two conditions together ensure that our stepping process is well-founded: if we step along edges in an acyclic graph and never turn around and step back, then we must eventually arrive at some vertex with only one adjacent edge, from which we just came, and we are then forced to stop.

Thus, in order to prove Theorem 2.5.2, it is sufficient to come up with a stepping strategy, and show that it satisfies these two conditions. This strategy, roughly speaking, works as follows:

- If we are currently at a thread i with $\rho(i) = \mathsf{Thread}(e)$, then by well-typedness of the expression e we can show that the thread can either take a step, or it is currently attempting to execute a lock operation on some lock $\langle k \rangle$. In the former case, we are done. In the latter case, we have i waiting$_\rho$ k, so we step to vertex k, and continue building our sequence of transitive waiting dependencies from there.

- If we are currently at a lock k with $\rho(k) = \mathsf{Lock}(\mathit{refcnt}, v)$, we can show, from the invariant we maintain about locks, that we are in one of the following situations:

  1. We have $v = \mathbf{None}$ and there is an incoming reference $\langle k \rangle$ from some $j \in \mathrm{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau_1^a \rangle$, *i.e.,* an opened reference.

  2. We have $v = \mathbf{Some}(v')$ and $\mathit{refcnt} = 0$, and there an incoming reference $\langle k \rangle$ from some $j \in \mathrm{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau_0^1 \rangle$, *i.e.,* a closed owner reference.

  3. We have $v = \mathbf{Some}(v')$ and $\mathit{refcnt} \neq 0$, and there an incoming reference $\langle k \rangle$ from some $j \in \mathrm{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau_0^0 \rangle$, *i.e.,* a closed client reference.

  In each case, if j is a lock, then we are immediately done because we can show i waiting$_\rho$ j and step to j. If j is a thread, then we have two cases:

  - The thread is waiting for us, *i.e.,* trying to do a lock operation with $\langle k \rangle$. In this case, the above information guarantees that this lock operation can proceed:

    1. In the first case with $v = \mathbf{None}$, we know that the only lock operations that are allowed by the type $\langle k \rangle : \mathbf{Lock}\langle \tau_1^a \rangle$ are **release** and **fork**, both of which can proceed. In particular, since the lock is open, the **wait** operation, which could block, is not permitted.

    2. In the second case with $v = \mathbf{Some}(v')$ and $\mathit{refcnt} = 0$, we have a closed owner reference, so the only potentially blocking operation that is permitted is **wait**, which can proceed since $\mathit{refcnt} = 0$.

    3. In the third case with $v = \mathbf{Some}(v')$ and $\mathit{refcnt} \neq 0$, we have a closed client reference, so none of the operations permitted are blocking.

  - The thread is not waiting for us, *i.e.,* we are waiting for the thread. So we step to the thread, and continue building our sequence of transitive waiting dependencies from there.

This concludes the sketch of the proofs of Theorem 2.5.1 and Theorem 2.5.2, which together can be used to prove our main Theorem 2.4.13, from which all the other

theorems in Section 2.4 follow. Although the description of the proofs here omit many details, particularly with respect to the preservation of acyclicity of the sharing topology, the description is nevertheless faithful to how the mechanized Coq proof is done.

2.6 THE $\lambda_{\text{LOCK++}}$ LANGUAGE: SHARING MULTIPLE LOCKS WITH LOCK GROUPS

The language λ_{lock} we have seen so far only allows us to share *one* lock with a child thread when we fork. To alleviate this restriction we now develop $\lambda_{\text{lock++}}$, which allows us to share more than one lock with a child thread. This allows us to handle locally cyclic connections.

The mechanism by which $\lambda_{\text{lock++}}$ allows this is *lock groups*. The locks of λ_{lock} store one pair (refcnt, **None** | **Some**(v)) of a reference count and an optional value, whereas the lock groups of $\lambda_{\text{lock++}}$ store a collection of such pairs. Whereas the type of locks in λ_{lock} is **Lock**$\langle \tau_b^a \rangle$, the type of a lock group in $\lambda_{\text{lock++}}$ is:

$$\textbf{Lock}_\textbf{G}\langle \tau_{1\,b_1}^{\,a_1}, \ldots, \tau_{n\,b_n}^{\,a_n} \rangle$$

That is, we generalize the data within the brackets from one item to n items, but each item still consists of a triple τ_b^a where τ indicates the type of that lock, a indicates whether we are the owner of that lock, and b indicates whether we have currently acquired that lock.

We are allowed to freely create and destroy empty lock groups:

$$\textbf{newgroup}_\textbf{G} : \mathbf{1} \to \textbf{Lock}_\textbf{G}\langle\rangle$$
$$\textbf{dropgroup}_\textbf{G} : \textbf{Lock}_\textbf{G}\langle\rangle \to \mathbf{1}$$

Once we have a lock group, we are able to create a new lock in the lock group, choosing at which position i to place it in the list:

$$\textbf{newlock}_\textbf{G}[i] : \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \vec{g}_2 \rangle \to \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \tau_1^1, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

Similarly, we are able to drop a lock from the group, provided it is a client reference:

$$\textbf{droplock}_\textbf{G}[i] : \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \tau_0^0, \vec{g}_2 \rangle \to \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

The more interesting operation is acquire, which acquires one of the locks in the group:

$$\textbf{acquire}_\textbf{G}[i] : \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \tau_0^a, \vec{g}_2 \rangle \to \textbf{Lock}_\textbf{G}\langle \vec{g}_1, \tau_1^a, \vec{g}_2 \rangle \times \tau \qquad \text{where } i = |\vec{g}_1| \text{ and } \textbf{closed } \vec{g}_2$$

Acquire requires that we obey the lock order, that is, we cannot **acquire** a lock in the group if there is an opened lock to its right in the type-level list. This condition is necessary to prevent **acquire-acquire** deadlocks. The rule for release is as folows:

$$\textbf{release}_G[i] : \textbf{Lock}_G\langle \vec{g}_1, \tau_1^a, \vec{g}_2 \rangle \times \tau \rightarrow \textbf{Lock}_G\langle \vec{g}_1, \tau_0^a, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

The condition for wait has to be even more stringent than the rule for acquire:

$$\textbf{wait}_G[i] : \textbf{Lock}_G\langle \vec{g}_1, \tau_0^1, \vec{g}_2 \rangle \rightarrow \textbf{Lock}_G\langle \vec{g}_1, \vec{g}_2 \rangle \times \tau$$
$$\text{where } \textbf{closed } \vec{g}_1, \vec{g}_2, \textbf{owners } \vec{g}_2 \text{ and } i = |\vec{g}_1|$$

The condition says that we can only wait if all locks are closed (to prevent **acquire-wait** deadlocks on different locks in the group), and we must obey the lock order with respect to the owners, that is, we cannot **wait** if there is a client to the right (to prevent **wait-wait** deadlocks on different locks).

The rule for fork allows us to share an entire lock group with the child thread:

$$\textbf{fork}_G : \textbf{Lock}_G\langle \vec{g} \rangle \times (\textbf{Lock}_G\langle \vec{g}_1 \rangle \multimap 1) \rightarrow \textbf{Lock}_G\langle \vec{g}_2 \rangle \qquad \text{where } \textbf{split } \vec{g} \textbf{ into } \vec{g}_1, \vec{g}_2$$

The relation **split $\vec{g}$ into $\vec{g}_1, \vec{g}_2$** specifies that locks are split as in λ_{lock}, but elementwise for each lock:

$$\tau_{b_1+b_2}^{a_1+a_2} \in \vec{g} \quad \text{is split into} \quad \begin{cases} \tau_{b_1}^{a_1} \in \vec{g}_1 \\ \tau_{b_2}^{a_2} \in \vec{g}_2 \end{cases}$$

The rules for $\lambda_{\text{lock++}}$ are summarized in Figure 15. In summary, $\lambda_{\text{lock++}}$ organises locks into lock groups, which can be grown and shrunk dynamically.

2.6.1 *Examples of Using Lock Orders*

The key improvement over λ_{lock} is that $\lambda_{\text{lock++}}$'s **fork$_G$** allows us to share an entire lock group, with potentially multiple locks:

$$\textbf{let } \ell = \textbf{newgroup}_G() \textbf{ in}$$
$$\textbf{let } \ell = \textbf{newlock}_G[o](\ell) \textbf{ in}$$
$$\textbf{let } \ell = \textbf{newlock}_G[1](\ell) \textbf{ in}$$
$$\textbf{let } \ell = \textbf{fork}_G(\ell, \lambda\ell. \cdots) \textbf{ in} \cdots$$

In a $\lambda_{\text{lock++}}$ version of the bank example (Section 2.2.3), this would allow us to have multiple bank threads that each do transactions over multiple locks, guaranteeing deadlock freedom because the type system ensures that the banks acquire the locks according to the lock order.

$$\frac{\Gamma \text{ unr}}{\Gamma \vdash \textbf{newgroup}_G() : \textbf{Lock}_G\langle\rangle} \qquad \frac{\Gamma \vdash e : \textbf{Lock}_G\langle\rangle}{\Gamma \vdash \textbf{dropgroup}_G(e) : \mathbf{1}}$$

$$\frac{\Gamma \vdash e : \textbf{Lock}_G\langle\vec{g}_1, \vec{g}_2\rangle \qquad i = |\vec{g}_1|}{\Gamma \vdash \textbf{newlock}_G[i](e) : \textbf{Lock}_G\langle\vec{g}_1, \tau_1^1, \vec{g}_2\rangle} \qquad \frac{\Gamma \vdash e : \textbf{Lock}_G\langle\vec{g}_1, \tau_0^0, \vec{g}_2\rangle \qquad i = |\vec{g}_1|}{\Gamma \vdash \textbf{droplock}_G[i](e) : \textbf{Lock}_G\langle\vec{g}_1, \vec{g}_2\rangle}$$

$$\frac{\Gamma \vdash e : \textbf{Lock}_G\langle\vec{g}_1, \tau_0^1, \vec{g}_2\rangle \qquad \textbf{closed } \vec{g}_1, \vec{g}_2, \textbf{owners } \vec{g}_2 \qquad i = |\vec{g}_1|}{\Gamma \vdash \textbf{wait}_G[i](e) : \textbf{Lock}_G\langle\vec{g}_1, \vec{g}_2\rangle \times \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \textbf{Lock}_G\langle\vec{g}\rangle \qquad \Gamma_2 \vdash e_2 : \textbf{Lock}_G\langle\vec{g}_1\rangle \multimap \mathbf{1} \qquad \textbf{split } \vec{g} \textbf{ into } \vec{g}_1, \vec{g}_2}{\Gamma \vdash \textbf{fork}_G(e_1, e_2) : \textbf{Lock}_G\langle\vec{g}_2\rangle}$$

$$\frac{\Gamma \vdash e : \textbf{Lock}_G\langle\vec{g}_1, \tau_0^a, \vec{g}_2\rangle \qquad i = |\vec{g}_1| \qquad \textbf{closed } \vec{g}_2}{\Gamma \vdash \textbf{acquire}_G[i](e) : \textbf{Lock}_G\langle\vec{g}_1, \tau_1^a, \vec{g}_2\rangle \times \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \textbf{Lock}_G\langle\vec{g}_1, \tau_1^a, \vec{g}_2\rangle \qquad \Gamma_2 \vdash e_2 : \tau \qquad i = |\vec{g}_1|}{\Gamma \vdash \textbf{release}_G[i](e_1, e_2) : \textbf{Lock}_G\langle\vec{g}_1, \tau_0^a, \vec{g}_2\rangle}$$

Figure 15: $\lambda_{\text{lock++}}$'s lock group typing rules.

Dining philosophers

We can use lock groups for a dynamic version of Dijkstra's dining philosophers (*a.k.a.,* a unbounded process network (Giachino et al., 2014; Kobayashi and Laneve, 2017)), where the number n of philosophers is chosen dynamically.

$$\text{dine} : \textbf{Lock}_G\langle\textbf{int}_0^{a_1}, \textbf{int}_0^{a_2}\rangle \multimap \mathbf{1}$$
$$\text{dine}(\ell) \triangleq$$
$$\quad \textbf{let } \ell, x = \textbf{acquire}_G[0](\ell) \textbf{ in}$$
$$\quad \textbf{let } \ell, y = \textbf{acquire}_G[1](\ell) \textbf{ in}$$
$$\quad \textbf{let } \ell = \textbf{release}_G[0](\ell, y) \textbf{ in}$$
$$\quad \textbf{let } \ell = \textbf{release}_G[1](\ell, x) \textbf{ in } \text{dine}(\ell)$$

$$\text{phil} : \textbf{int} \times \textbf{Lock}_G\langle\textbf{int}_0^{a_1}, \textbf{int}_0^{a_2}\rangle \multimap \mathbf{1}$$
$$\text{phil}(0, \ell) \triangleq \text{dine}(\ell)$$
$$\text{phil}(n+1, \ell) \triangleq$$
$$\quad \textbf{let } \ell = \textbf{release}_G[2](\textbf{newlock}_G[2](\ell), 42) \textbf{ in}$$
$$\quad \textbf{let } \ell = \textbf{fork}_G(\ell, \lambda\ell. \text{dine}(\textbf{droplock}_G[0](\ell))) \textbf{ in}$$
$$\quad \text{phil}(n, \textbf{droplock}_G[1](\ell))$$

We can start the philosophers by running the following code:

$$\textbf{let } \ell = \textbf{newgroup}_G() \textbf{ in}$$
$$\textbf{let } \ell = \textbf{release}_G[0](\textbf{newlock}_G[0](\ell), 42) \textbf{ in}$$
$$\textbf{let } \ell = \textbf{release}_G[1](\textbf{newlock}_G[1](\ell), 42) \textbf{ in}$$
$$\textbf{let } \ell = \textbf{fork}_G(\ell, \lambda\ell.\ \text{dine}(\ell)) \textbf{ in } \text{phil}(n, \ell)$$

The code sets up a ring of forks (the locks) and philosophers (the dining threads) between them. It helps to think of the ring as a long line of forks, which is closed by making a philosopher dine with the first and last forks in the line. Intuitively, the phil function takes a lock group with two forks: the very first fork in the line, and the last fork in the line so far. The function then creates a new fork at the end of the line, and makes a new philosopher dine with the last two forks. We then forget about the penultimate fork, and make a recursive call. At the end of the recursion, we close the loop by making a philosopher dine with the very first fork and the last. To initialize this process, we create the first two forks in the line. We make a philosopher dine with these forks, and use phil to make the line and close the loop. Note that there will be $n + 2$ locks in the lock group overall, but the local view of any reference has at most 3 locks visible at any time.

Growing the table

To illustrate the dynamic nature of lock groups and their lock orders, we can modify the above code to dynamically grow the number of philosophers at the circular table. To do so, replace the recursive call of dine(ℓ) with phil($2, \ell$), making dine and phil mutually recursive. Now, after a philosopher is done dining (*i.e.,* acquiring and releasing their two locks), the philosopher replaces itself with 3 dining philosophers, thus growing the circle by 2.

Growing the table, fractally

To further illustrate the dynamic nature, consider replacing the recursive call dine(ℓ) with the following code:

$$\text{phil}(2, \textbf{fork}_G(\ell, \lambda\ell.\ \text{phil}(2, \ell)))$$

   After dining, the philosopher replaces itself with 6 philosophers, arranged in two parallel lines. The leftmost philosophers in the two lines both use the left-hand lock of the original philosopher, and the rightmost philosophers in the two lines both use the right-hand lock of the original philosopher.
   Of course, each of these philosophers is running the same code, and after dining they will in turn replace themselves with such parallel lines, according to the following substitution:
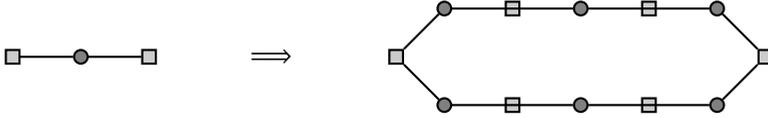
In this picture, the squares are forks (locks) and the circles are philosophers (threads). Some forks are now accessed by more than 2 philosophers, and the number of philosophers accessing a fork can grow dynamically. Which philosopher dines and splits is non-deterministic, and thus the initial table of 42 grows non-deterministically in fractal and intricately interconnected circular patterns. Nevertheless, the type system of $\lambda_{\text{lock++}}$ guarantees deadlock freedom by construction.

Multiple lock groups

Note that so far we have used only a single lock group. The expressiveness of λ_{lock} for multiple locks based on the sharing topology is still available for $\lambda_{\text{lock++}}$, but now for multiple lock groups. The reader may wonder how the expressivity of pure lock orders with a single lock group of $\lambda_{\text{lock++}}$ compares with the expressivity of the pure sharing topology of λ_{lock}. The two mechanisms are orthogonal, and one is not strictly more powerful than the other, because two locks in the same lock group always need to be locked in the given order, whereas two independent locks can be locked in any order.

2.6.2 *References to Lock Groups*

Operationally, $\lambda_{\text{lock++}}$ works precisely like λ_{lock}, except that each lock group now stores a collection of locks, each of which is identified by an id (a natural number). Each reference to a lock group may only have partial knowledge of which locks are present in the group, because new locks may have been added concurrently by other threads that hold a reference to the same lock group. However, note that the operations **newlock**$_G$[i], **droplock**$_G$[i], **acquire**$_G$[i] and **release**$_G$[i] refer to a lock by index i, which is the index of the *local view* of the lock group. Therefore, each reference to a lock group now consists not just of $\langle k \rangle$, but in fact of $\langle k \mid i_0, i_1, \ldots, i_n \rangle$, where k identifies the lock group, and $i_0, i_1, \ldots, i_n$ identifies *which* locks in the lock group this reference knows about. Thus, when we have $\langle k \mid i_0, i_1, \ldots, i_n \rangle : \textbf{Lock}_G \langle \vec{g} \rangle$, we have $|\vec{g}| = n$.

2.6.3 *The Invariant for Lock Groups*

The invariant for lock groups is very similar to the one for locks. The sharing topology does not distinguish between the individual locks in a group, but treats them as an atomic whole. Thus, we may have edges between threads and lock groups, and between lock groups and lock groups, and this graph must be acyclic. The local invariant for a lock group with respect to the types $\langle k \mid i_0, i_1, \ldots, i_n \rangle : \textbf{Lock}_G \langle \vec{g} \rangle$

of all the references to the lock group, is also similar. Elementwise, we insist the same as for single locks: each lock must have precisely one owning reference, and the reference count of each lock must agree with the number of client references. Furthermore, whether the lock stores **None** | **Some**(v) must agree with the existence of an open reference. In other words, the invariant for a lock group is the same as for single locks, but elementwise.

The key difference is that for lock groups, we insist that *the order of the lists* $I = i_0, i_1, \ldots, i_n$ of the lock references of the various references *must agree*. That is, there is some list I_{all} such that the I of every client reference $\langle k \mid I \rangle$ to the lock group is a subsequence of it.

This invariant is preserved by all the operations:

- Inserting a new lock into the group inserts it into I_{all} as well.

- Deleting a lock using **wait**$_G$[i] deletes it from I_{all} as well.

- Dropping a client reference to a lock has no effect on I_{all}; we only need to ajust the subsequence witness of that reference.

- Acquiring and releasing a lock has no effect on I_{all}.

- Forking a lock group has no effect on I_{all}, since the two newly split references have the locks in the same order as the original reference.

In summary, the references to the lock group have a partial but consistent view of the lock order.

2.6.4 *Reachability for Lock Groups*

We wish to generalize the theorems of Section 2.4 to $\lambda_{\text{lock++}}$. This turns out to be very easy: only the definition of blocked (Theorem 2.4.2) depended on the details of λ_{lock}. We adjust it so that a thread is considered blocked on a lock group if it is trying to perform one of $\lambda_{\text{lock++}}$'s lock group operations on it.

In Section 2.5 we have seen that the difficult case of the reachability proof is to establish reachability of the locks, or in our case now, a lock group. The reachability proof for a lock comes down to showing that it is impossible that every reference to the lock is blocked on it. Let us thus see why this also holds for lock groups. Suppose that there is some reference to the lock group. If this reference is doing anything other than acquire or wait on this lock group, then it is not blocked, and hence we're done. If it is blocked, we have the following two cases:

The case of acquire

Consider the case when this reference is doing an acquire of some lock i in the group. If this acquire can't proceed, the lock must have already been acquired, via some other reference. Consider, now, the thread holding that reference (if it is held

by a lock, we are also immediately done). If that thread is not blocked on this lock group, we're done. If it *is* blocked on this lock group, it could be doing an acquire, or a wait. In fact, it is not possible that the thread is doing a wait operation, because the typing rule of wait says that *all the locks must be closed*, and if it has acquired a lock in the group, this condition is violated. Hence, the thread must be doing an acquire of some lock j in the group. It seems that we're now back to where we started. However, due to the typing rule of acquire, the lock j must be higher in the lock order than lock i. Thus, we have made progress, in the sense that we have gone up in the lock order. Hence, if we keep repeating this reasoning, we go up and up in the lock order, until we're at the end, and then the thread can't be doing any acquire (formally, we phrase this using induction). In summary, if some thread is doing an acquire, then there must be some reference into the lock group that is not blocked on the lock group, and we're done.

The case of wait

Now consider the case when the reference is doing a wait on some lock i in the group. If this wait cannot proceed, then the refcount of that lock must be nonzero, so there must also be some client reference to i. Consider, now, the thread holding that reference (if it is held by a lock, we are also immediately done). If that thread is not blocked on this lock group, we're done. If it *is* blocked on this lock group, it could be doing an acquire, or a wait. If it's doing an acquire, we're done, by the preceding paragraph. Hence, suppose that the thread is doing a wait on some lock j in the group. It seems that we're now back to where we started. However, due to the typing rule of wait, the lock j must be higher in the lock order than lock i. Hence, by repeating this reasoning, we go up in the lock order, and eventually we're done. In summary, if some thread is doing a wait, then there must be some reference into the lock group that is not blocked on the lock group, and we're done.

2.7 MECHANIZED PROOFS

All of our theorems (Theorem 2.4.9, Theorem 2.4.12, Theorem 2.4.13, Theorem 2.5.1, and Theorem 2.5.2) have been mechanized in the Coq proof assistant (Coq Team, 2021), for both λ_{lock} (⚙) and $\lambda_{\text{lock++}}$ (⚙). The mechanization is structured as follows:

- The $\lambda_{\text{lock}}/\lambda_{\text{lock++}}$ language definition: expressions, static type system (with unrestricted and recursive types), and operational semantics.

- The configuration invariant, stating that the configuration remains well typed, that the sharing topology is acyclic, and that the lock invariants hold for every lock / lock group.

- Proof that the invariant is preserved by the operational semantics (Theorem 2.5.1).

- Proof that configurations satifsying the invariant are fully reachable (Theorem 2.5.2).

- Proofs that full-reachability is equivalent to deadlock/leak freedom, and that they imply type safety and global progress (Theorem 2.4.9, Theorem 2.4.12).

In order to handle recursive types, we use the coinductive method of Gay et al. (2020). The mechanization uses a graph library to reason about the graph underlying the sharing topology (Jacobs et al., 2022b) and depends on Iris, mainly for the Iris proof mode (Jung et al., 2015; Krebbers et al., 2017b; Jung et al., 2018b), as well as on the stdpp extended standard library for Coq (Coq-std++ Team, 2021).

2.8 RELATED WORK

Related work on deadlock freedom spans both shared memory and message-passing concurrency as well as type systems and program logics. Related work on memory leak freedom seems to be confined to the purely linear setting. While memory safety has been studied both in research (Tofte and Talpin, 1997; Grossman et al., 2002) and in practice, with Rust as the most prominent example (Jung et al., 2018a), memory safety does not entail memory leak freedom.

SESSION TYPES   Conceptually, our work is most closely related to the family of binary session types (Toninho et al., 2013; Toninho, 2015; Lindley and Morris, 2015; Caires et al., 2016; Lindley and Morris, 2016c, 2017; Fowler et al., 2019; Kokke et al., 2019; Fowler et al., 2021; Jacobs et al., 2022b) that build on the Curry-Howard correspondence between linear logic and the session-typed π-calculus (Caires and Pfenning, 2010; Wadler, 2012). Like these systems, our type system uses linearity to restrict the propagation of references to rule out circular waiting dependencies among a program's run-time objects. Our **fork** construct, moreover, resembles process spawning (*a.k.a.,* cut) in that it connects a parent and a child thread with exactly one lock (group). However, our **fork** construct differs in that it allows a parent thread to *share* the same lock (group) among repeatedly forked off children (*e.g.,* example in Section 2.2.3), permitting aliases to a lock (group) to exist and threads with such aliases to affect each other. We remark that the linear exponential, supported by some linear session types, has a copying and not a sharing semantics.

Traditional session types, both binary (Honda, 1993; Honda et al., 1998) and multiparty (Honda et al., 2008), suffer from deadlock. Carbone and Debois (2010) were the first to explore the benefits of acyclicity of the underlying communication topology for deadlock freedom. These ideas, combined with insights gained from the Curry-Howard correspondence between linear logic and the session-typed π-calculus (Caires and Pfenning, 2010; Wadler, 2012), gave rise to a series of work to establish deadlock freedom for multiparty session types (Carbone et al., 2015, 2016, 2017; Castro-Perez et al., 2021; Jacobs et al., 2022b). While our notion of sharing topology

draws inspiration from these works, our type system offers unrestricted sharing through locks.

Recently, Qian et al. (2021) and Rocha and Caires (2021) have inhaled the linear exponential a slightly different semantics. In particular, Qian et al. (2021) observe that the established interpretation of the linear exponential (Wadler, 2012) fails to faithfully encode client-server interactions, where clients are served in a non-deterministic fashion. The authors complement the linear exponential with a coexponential to fill this gap. Like acquires in our language, Qian et al. (2021)'s coexponential is the source of non-determinism. However, the coexponential still has a copying semantics, ruling out various sharing scenarios, such as dining philosophers (see Section 2.6.1).

Closer to our base calculus λ_{lock} is Rocha and Caires (2021)'s PaT language with reference cells. PaT's reference cells have constructs for reading a cell's contents, updating it, and locking it, while remaining deadlock-free. To account for the non-determinism resulting from an update, the authors introduce non-deterministic sums from differential linear logic (Ehrhard and Regnier, 2006; Ehrhard, 2018). A similarity between PaT and λ_{lock} is the reliance on acyclicity for deadlock freedom not just for sessions, but also for reference cells. PaT ensures acyclicity with co-contraction rules for its **share** construct, which serves a similar purpose as λ_{lock}'s **fork**, with the difference that PaT is situated in a π-calculus, rather than a λ-calculus like λ_{lock}. Like **fork**, the typing rules of **share** distribute the lock's open/closed state over parallel processes, ensuring that only a single process is interacting with an opened lock. In contrast to PaT's reference cells, which can only store unrestricted values (*i.e.,* values that can be freely copied and discarded, such as natural numbers), λ_{lock}'s locks can store arbitrary linear values, including values representing non-affine obligations. Locking and unlocking transfers full ownership over the contents, including obligations, such as the obligation to close a lock or send a message on a channel. In terms of our invariants (Section 2.2, principles 1-5, and Section 2.5), these obligations emerge as edges between two locks as well as the need to introduce the owner/client distinction in addition to the open/closed distinction. Moreover, our extended language $\lambda_{\text{lock++}}$ supports cyclic sharing topologies, which are beyond the reach of reference cells.

Among the extensions of linear logic session types with non-determinism and notions of sharing, manifest sharing (Balzer and Pfenning, 2017; Balzer et al., 2018, 2019) is the work closest to ours. Manifest sharing introduces an adjoint formulation of linear and shared session types, with the latter resembling our locks, which can be freely shared and must be communicated with by entering a critical section. Mutual exclusion is enforced by adjoint modalities, with an acquire and release semantics. While the original system (Balzer and Pfenning, 2017; Balzer et al., 2018) can suffer from deadlocks, Balzer et al. (2019) augment manifest sharing with partial orders to rule out deadlocks. In contrast to our locks, shared processes in (Balzer et al., 2019) cannot store any linear resources. Moreover, while Balzer et al. (2019)'s system supports order-polymorphic processes, ensuring compositionality, local orders must

comply with a global order at run-time, whereas lock group orders in $\lambda_{\text{lock++}}$ are independent of each other. Lastly, Balzer et al. (2019)'s system does not support unbounded process networks (see Section 2.6.1), whereas $\lambda_{\text{lock++}}$ does.

USAGES AND OBLIGATIONS   The addition of channel usage information to types in a concurrent, message-passing setting was pioneered by Kobayashi (1997); Igarashi and Kobayashi (1997), who applied the idea to deadlock prevention in the π-calculus as well as race freedom (Igarashi and Kobayashi, 2001, 2004). Typically, types are augmented with the relative ordering of channel actions, with the type system ensuring that the transitive closure of such orderings forms a strict partial order. Building on this, Kobayashi (2002a) proposed type systems that ensure a stronger property, dubbed lock freedom, and variants that are amenable to type inference (Kobayashi et al., 2000; Kobayashi, 2005). Kobayashi (2006) extended this to account for recursive processes and type inference. The most advanced system (Giachino et al., 2014; Kobayashi and Laneve, 2017) in this series supports unbounded process networks, allowing dynamic creation of circular topologies.

Padovani (2014) contributes a simplified account of Kobayashi-style orders, albeit at the cost of expressivity, which, assuming linear channel usage, gets by with a single priority rather than usage information. Padovani (2014)'s system supports priority polymorphism to support cyclic interleavings of recursive processes. Padovani's system also served as a source of inspiration for the development of a functional language with session types by Dardha and Gay (2018); Kokke and Dardha (2021c). The authors' system focuses on the integration with a functional language, and currently lacks support of recursive circular behavior. Kobayashi-style orders have also been adopted in the multiparty session type setting (Coppo et al., 2013; Bettini et al., 2008; Coppo et al., 2016) to establish global progress in the presence of multiparty session interleavings.

Like Giachino et al. (2014); Kobayashi and Laneve (2017)'s system, our extended system $\lambda_{\text{lock++}}$ supports unbounded process networks (see Section 2.6.1). However, the two systems differ conceptually: whereas our approach is primarily guided by topology, Kobayashi-style orders are guided by orders. As a result, the systems differ in technical details and user experience. For example, our core language λ_{lock} does not require any additional annotations, deadlock freedom simply follows from thread-local linearity. On the other hand, $\lambda_{\text{lock++}}$ does require lock orders. These orders, however, are purely local to a lock group, and there is no need for local orders to comply with each other or a global lock order, or any other condition across groups, when acquiring locks from distinct groups. To the best of our knowledge, this feature is novel and increases compositionality.

PROGRAM LOGICS   Our work is tangentially related to works using Hoare logics with lock orders (Leino et al., 2010; Hamin and Jacobs, 2018) for deadlock freedom and work using concurrent separation logic (da Rocha Pinto et al., 2014; Jung et al., 2015, 2018b; Nanevski et al., 2019; D'Osualdo et al., 2021a; Farka et al., 2021) for

program verification. Our focus is on type-based, and thus automated verification. A fully fledged separation logic that is capable of proving deadlock freedom using sharing topologies is still missing. This is something we hope to explore in the future.

2.9 LIMITATIONS AND FUTURE WORK

No decidable type system is without its limitations, and $\lambda_{\text{lock++}}$ is no exception. To our understanding, the main limitations of λ_{lock} and $\lambda_{\text{lock++}}$ are as follows:

Lock group references have static size.

While locks can be added to a lock group dynamically, controlled by a run-time variable n (e.g., in dynamic dining philosophers), the number of locks that can be accessed *via a single lock group reference* at any given point in the program is statically determined by the length of the type-level list (*e.g.,* in dining philosophers, each philosopher accesses two locks). This type dependency curtails expressivity when a lock group reference is used in a loop, requiring the type to be invariant across iterations and thus fixing simultaneous access to a statically predetermined number of locks in a lock group, rather than adjusting that number dynamically.

DAG-shaped mutable data structures.

The simple locks of λ_{lock} can be used as mutable reference cells. The operational semantics employs reference counting memory management. Reference counting guarantees memory leak freedom as long as the data has the shape of a directed acyclic graph (DAG), and is often used that way. In a DAG, as opposed to a tree, a node may have multiple parents. In λ_{lock}, a node can have multiple parents as well, but these parents are always *disjoint*. Thus, in terms of data shapes that can be expressed, λ_{lock} supports a strict superset of tree-shapes, but a strict subset of DAG-shapes. It would be nice to relax this restriction and support general DAGs, but we do not know how to do so without introducing the possibility of deadlock. The issue is that we could potentially obtain a duplicate reference to the same lock via different paths in the DAG, which can be used to create deadlocks.

Rust's `unsafe`

Rust has the `unsafe` mechanism for code that violates the rules of the borrow checker, to be used if the programmer promises that the code is safe and upholds Rust's invariants (Jung et al., 2018a). One could analogously imagine an `unsafe` construct for λ_{lock} that allows the programmer to violate the linearity restriction and freely duplicate a lock, if the programmer promises that the code is safe and upholds λ_{lock}'s invariants. The open problem would be characterizing the invariants that the programmer would be responsible for upholding in their unsafe code, such that

deadlock and leak freedom is guaranteed even when their unsafe code is mixed with other code. A mechanized meta theory will require extending tools like Iris with support for deadlock and leak freedom based on sharing topologies.

2.10 CONCLUSION

We have presented λ_{lock}, a language with locks where deadlock and memory leak freedom is guaranteed by type checking. Deadlock and leak freedom are ensured by restricting the sharing topology between locks and threads. This enables the λ_{lock} type system to be free of additional checks, such as lock orders.

The locks in λ_{lock} are *higher-order*, meaning that we can store arbitrary linear values in locks, and locks themselves are completely first class entities. In particular, we can store locks in locks. This is a crucial ingredient that allows us to implement session-typed channels in terms of locks.

We have also presented $\lambda_{\text{lock++}}$, which extends λ_{lock} with *lock groups*, and allows sharing multiple locks with the child thread when we fork. This is kept deadlock free by requiring the **acquire** and **wait** operations to happen in accordance with the lock order local to the group. Crucially, there is no global order, and different lock groups can be interacted with completely independently; locks from different lock groups can be acquired simultaneously without restrictions.

We hope that this is a step toward the end goal of having an expressive concurrent language where deadlock and leak freedom follow from the type system. As a next step toward this, we would like to distill more general principles that allow us to design concurrent languages based on the sharing topology.

Chapter 3

Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom

ABSTRACT   Session types have recently been integrated with functional languages, bringing message-passing concurrency to functional programming. Channel endpoints then become first-class and can be stored in data structures, captured in closures, and sent along channels. Representatives of the GV (Wadler's "Good Variation") session type family are of particular appeal because they not only assert session fidelity but also deadlock freedom, inspired by a Curry-Howard correspondence to linear logic. A restriction of current versions of GV, however, is the focus on binary sessions, limiting concurrent interactions within a session to two participants. This chapter introduces Multiparty GV (MPGV), a functional language with multiparty session types, allowing concurrent interactions among several participants. MPGV upholds the strong guarantees of its ancestor GV, including deadlock freedom, despite session interleaving and delegation. MPGV has a novel redirecting construct for modular programming with first-class endpoints, thanks to which we give a type-preserving translation from binary session types to MPGV to show that MPGV is strictly more general than binary GV. All results in this chapter have been mechanized using the Coq proof assistant.

3.1   INTRODUCTION

Session types are a type discipline for message-passing concurrency. Originally developed in the context of process calculi by Honda (1993); Honda et al. (1998), they were later generalized to object-oriented (Dezani-Ciancaglini et al., 2006) and functional languages (Gay and Vasconcelos, 2010) leading to implementations in mainstream languages like Haskell (Pucella and Tov, 2008; Imai et al., 2010; Lindley and Morris, 2016b), Scala (Scalas and Yoshida, 2016b), OCaml (Padovani, 2017; Imai et al., 2019), and Rust (Jespersen et al., 2015a; Kokke, 2019; Chen et al., 2022).

A particularly exciting development is the GV ("Good Variation") session type family, pioneered by Gay and Vasconcelos (2010), later coined GV and refined by Wadler (2012), and further developed by *e.g.,* Lindley and Morris (2015, 2016c, 2017); Fowler et al. (2019, 2021); Kokke and Dardha (2021c); Jacobs et al. (2022b). The GV family combines session types with functional programming by treating session-typed channels as first-class data, similar to references in ML. Channels can be stored in data structures (like lists), captured by closures, and sent along channels (even when contained in data structures, thus generalizing delegation). Similarly, the

SILL family of session type languages (Toninho et al., 2013; Pfenning and Griffith, 2015; Toninho, 2015) integrates a process language via a contextual monad with an unrestricted functional language.

Aside from a tight integration with functional programming, a key strength of GV and SILL representatives is that they do not only guarantee type safety ("well-typed programs cannot get stuck due to illegal operations"), but also deadlock freedom or global progress ("well-typed programs cannot end up waiting for each other"). This result follows from adopting a session initialization pattern based on cut, inspired by the Curry-Howard correspondence between linear logic and the session-typed π-calculus (Caires and Pfenning, 2010; Wadler, 2012). Such a pattern *combines* session creation and thread spawning to avoid deadlocks. The family of session types based on the pioneering work by Honda (1993); Honda et al. (1998), in contrast, *separates* session creation from thread (process) spawning and thus does not prevent deadlocks. A cut-based initialization pattern also seamlessly integrates with channels as first-class data.

The restriction of interactions to *two* participants, present in GV, SILL, and session types based on the pioneering work by Honda (1993); Honda et al. (1998), led to the development of multiparty session types (Honda et al., 2008, 2016). Multiparty session types allow an arbitrary but statically determined number of participants ("roles") to engage in a session. The key ingredient of multiparty session types is a *global type* that defines a protocol from the perspective of the entire session, from which local types for each participant can be generated. A global type not only increases expressivity but also establishes deadlock freedom for a system consisting of a single session.

The development of GV-style session types and multiparty session types has mostly happened independently of each other. There exists no system that combines the flexibility of functional programming with the expressivity of multiparty session types. This chapter introduces **Multiparty GV (MPGV)**—a linear lambda calculus with first-class multiparty sessions and dynamic thread and channel initialization. Deadlock freedom is guaranteed purely by linear type checking and an n-ary "fork" inspired by a cut-based initialization pattern. MPGV complements linear sessions with standard unrestricted functional types and language features, such as general recursive functions and algebraic data types. The integration of multiparty session types into a GV-style functional language brings a number of challenges:

DEADLOCK FREEDOM. Although global types guarantee deadlock freedom for a single multiparty session, global types alone cannot guarantee deadlock freedom for interleaved multiparty sessions. To establish deadlock freedom in the presence of dynamic session spawning and session delegation, where participants can engage in several multiparty sessions simultaneously, Kobayashi-style "orders/priorities" (Kobayashi, 1997, 2002a, 2006) have been used to rule out cyclic dependencies among channel actions. The resulting interaction type systems (Coppo et al., 2013; Bettini et al., 2008; Coppo et al., 2016) are complementary in terms of expressivity compared

to GV. They are more powerful in the sense that they allow cyclic communication topologies within and between sessions. However, well-typed programs in GV cannot be translated into these systems because orders/priorities are static and sessions are not first-class data.

In this chapter we take the GV approach to deadlock freedom—MPGV features an n-ary "fork" that combines the creation of n threads and multiparty session for n participants. While this makes the MPGV type system and operational semantics simple, proving that it in fact guarantees deadlock freedom is challenging. To handle dynamic thread and channel creation, direct-style deadlock freedom proofs of GV (like those by Lindley and Morris (2015); Fowler et al. (2021); Jacobs et al. (2022b)) crucially rely on the communication topology remaining acyclic during program execution. For multiparty session types this is not the case—the communication topology *between* sessions is acyclic, but the communication topology *within* a session is not. The key insight of our work is to represent the cyclic communication topology within sessions as an acyclic graph at the logical level, without needing central coordination in the operational semantics.

PARTICIPANT REDIRECTING. Binary session types specify the types of data that is being sent and received, while local multiparty session types also specify the participants *names* to/from whom that data is received. These names make programming with first-class sessions non modular since the exact participants are fixed in type signatures. Suppose that one has library functions f and g such that f returns a session of a certain session type, and g expects an argument with that same session type, but with different participant names. We introduce a "redirecting" construct, which allows an endpoint to be passed to functions where different participant names are expected. Using this construct, we give a type-preserving translation from binary session types into MPGV, showing that MPGV restricted to two participants per session is at least as expressive as GV.

MECHANIZATION   The complexities of session types, especially in the multiparty setting, and the existence of published broken proofs—including the failure of subject reduction for several multiparty systems, as shown by Scalas and Yoshida (2019)—gave the impetus for mechanization. Whereas there exists extensive work on mechanizing the meta-theory of binary session types (Thiemann, 2019; Rouvoet et al., 2020; Hinrichsen et al., 2021; Tassarotti et al., 2017; Goto et al., 2016; Ciccone and Padovani, 2020; Castro-Perez et al., 2020; Gay et al., 2020), deadlock freedom for binary session types has only recently been mechanized by Jacobs et al. (2022b). For multiparty session types, the only mechanization is Zooid by Castro-Perez et al. (2021), which mechanizes the trace semantics of a *single* multiparty session and proves that it conforms to its global type. In the spirit of this line of work, we provide a full mechanization of all our results with the Coq proof assistant.

CONTRIBUTIONS AND OUTLINE Our main contribution is **MPGV**—the first deadlock-free linear lambda calculus with first-class multiparty sessions, dynamic thread and channel initialization, and functional features like general recursive functions and algebraic data types. Concretely:

- We explain the key ideas behind MPGV in the context of new and existing examples (Section 3.2).

- We formalize the type system and operational semantics of MPGV (Section 3.3).

- We give a type-preserving embedding of GV-style binary session types into MPGV, using our new redirecting construct, showing that MPGV goes strictly beyond binary session types (Section 3.4).

- We prove a combined partial deadlock and memory-leak freedom theorem for multiparty session types that also subsumes type safety and global progress (Section 3.5 and Section 3.7).

- Inspired by Scalas and Yoshida (2019), we extend MPGV with a more flexible notion of consistency that does not rely on global types (Section 3.6).

- We mechanize all our results in the Coq proof assistant (Section 3.8).

3.2 MPGV BY EXAMPLE

We introduce MPGV's features, based on examples (Section 3.2.1–Section 3.2.8), and provide the main intuitions for how MPGV guarantees deadlock freedom for cyclic intra-session topologies (Section 3.2.9).

3.2.1 *Global and Local Types*

Similar to original multiparty session types (Honda et al., 2008), sessions in MPGV can be described by a *global* type. A simple example of a global type is:[1]

$$G \triangleq [0 \to 1]\mathbf{N}.[1 \to 2]\mathbf{N}.[2 \to 0]\mathbf{N}.\mathsf{End}.$$

This global type says that participant 0 first sends a value of natural number type **N** to participant 1, then 1 sends a **N** to 2, then 2 sends a **N** to 0, and finally the protocol ends. The global type G induces *local types* for each participant p via projections G ↓ p:

G ↓ 0 = **![1]N.?[2]N**.End   G ↓ 1 = **?[0]N.![2]N**.End   G ↓ 2 = **?[1]N.![0]N**.End

---

[1] The term *global type* is a bit of a misnomer: it is only global to a given communication session between multiple participants, in contrast to a local type, which is local to a participant. A global type is not global to the whole program, because a program in MPGV can have multiple communication sessions.

The local type $![p]\tau.L$ indicates that the next action should be sending a value v of type τ to participant p, to then continue with L. Dually, $?[p]\tau.L$ indicates that the next action should be receiving a value v of type τ from participant p, to then continue with L. Finally, End states that the protocol has finished and the participant's endpoint should be closed.

3.2.2 *Combined Session and Channel Initialization*

With our simple global type G at hand, we now give a program that implements this global type:

> **let** $c_0 : ![1]N.?[2]N.End = \textbf{fork}(\mathit{service}_1, \mathit{service}_2)$ **in**
> **let** $c_0 : ?[2]N.End = \textbf{send}[1](c_0, 99)$ **in**
> **let** $(c_0, n) : End \times N = \textbf{receive}[2](c_0)$ **in**
> **close**(c_0)

The **fork** operation simultaneously forks off 2 threads and creates 3 channel endpoints for the participants in the session. The **fork** returns endpoint c_0 with type $G \downarrow 0 = ![1]N.?[2]N.End$, and runs functions $\mathit{service}_1$ and $\mathit{service}_2$ (shown below) in background threads. The main thread uses **send**$[1](c_0, 99)$ to send the message "99" to participant 1 (*i.e.*, $\mathit{service}_1$). As is common in functional session-typed languages, the **send** and **receive** operations of MPGV return the endpoint back to us. The returned endpoint will be at a different type, because the step has been taken in the session type. For convenience, the above code let-binds the returned endpoint to the same name. The main thread then uses the operation **receive**$[2](c_0)$ and blocks to receive a message from endpoint 2 (*i.e.*, $\mathit{service}_2$). After the message has been received, it closes the endpoint using **close**.

Similar to other multiparty session-type systems, MPGV uses natural numbers for participant names in **send** and **receive** to indicate which other participant the communication concerns. Note that also for **receive** it is necessary to indicate which participant to receive from, because multiple participants could send a message to the same participant simultaneously, and these messages may have different types (for instance, if participant 1 sends an integer to participant 0, and participant 2 sends a string to participant 0; these messages may arrive in any order because of asynchronous communication, so a receive operation on endpoint 0 has to specify which participant it is receiving from in order to have a sound type system). The endpoint returned from **fork** has participant number 0, and endpoints of the forked-

off threads have participant numbers 1, 2, *etc.* The forked-off threads could be implemented as:

$$service_1 : (\mathbf{?}[0]\mathbf{N}.\mathbf{!}[2]\mathbf{N}.\mathsf{End}) \rightarrow \mathbf{1}$$

$$service_1 \; c_1 \triangleq \mathbf{let} \; (c_1, n) = \mathbf{receive}[0](c_1) \; \mathbf{in}$$
$$\mathbf{let} \; c_1 = \mathbf{send}[2](c_1, n + 3) \; \mathbf{in}$$
$$\mathbf{close}(c_1)$$

$$service_2 : (\mathbf{?}[1]\mathbf{N}.\mathbf{!}[0]\mathbf{N}.\mathsf{End}) \rightarrow \mathbf{1}$$

$$service_2 \; c_2 \triangleq \mathbf{let} \; (c_2, n) = \mathbf{receive}[1](c_2) \; \mathbf{in}$$
$$\mathbf{let} \; c_2 = \mathbf{send}[0](c_2, n + 4) \; \mathbf{in}$$
$$\mathbf{close}(c_2))$$

The arguments of **fork** are closures that take the endpoint (typed with local type $G \downarrow p$) as argument and return the unit value when done. The first forked-off thread $service_1$ tries to receive a message from participant 0 (*i.e.,* the main thread), increments the received number, and passes it on to endpoint 2 (*i.e., $service_2$*). Similarly, the second forked-off thread $service_2$ receives a number from participant 1 (*i.e., $service_1$*), increments it, and passes it to participant 0 (*i.e.,* the main thread).

NOVEL ELEMENTS OF MPGV   The n-ary **fork** ensures that the communication topology between sessions remains acyclic. This is in contrast to original multiparty session-type systems (Honda et al., 2008), which use service names to create new sessions between already existing, concurrently running processes, selecting the participating processes non-deterministically in case there are several attempting to participate (see Section 3.9 for an in-depth discussion). By separating session creation from thread spawning in these original systems, cyclic communication topologies can be created, and hence interleaved sessions can deadlock. Inspired by binary session-typed lambda-calculi like GV (Wadler, 2012; Lindley and Morris, 2015) and multi-cut with coherence proofs (Carbone et al., 2015, 2016, 2017), MPGV combines session creation with thread spawning, to maintain acyclicity of the communication topology and guarantee deadlock freedom.

3.2.3 *Interleaving and First-Class Endpoints*

We now illustrate MPGV's support for session interleaving and delegation. Similar to the original versions of GV by Gay and Vasconcelos (2010); Wadler (2012), MPGV obtains delegation without the need for special language constructs since endpoints are first class. We modify the example from Section 3.2.2, which performs its

communication actions on c_0 locally, by letting the main thread fork off yet another thread to perform the communication:

$$\textbf{let } c_0 : G \downarrow 0 = \textbf{fork}(service_1, service_2) \textbf{ in}$$
$$\textbf{let } d_0 : G' \downarrow 0 = \textbf{fork}(\lambda d_1 : G' \downarrow 1.$$
$$\quad \textbf{let } (d_1, x) : (![0]\textbf{N}.\textbf{End}) \times (G \downarrow 0) = \textbf{receive}[0](d_1) \textbf{ in}$$
$$\quad \textbf{let } x : \textbf{?}[2]\textbf{N}.\textbf{End} = \textbf{send}[1](x, 99) \textbf{ in}$$
$$\quad \textbf{let } (x, n) : \textbf{End} \times \textbf{N} = \textbf{receive}[2](x) \textbf{ in}$$
$$\quad \textbf{let } d_1 : \textbf{End} = \textbf{send}[0](d_1, n) \textbf{ in}$$
$$\quad \textbf{close}(x); \textbf{close}(d_1)) \textbf{ in}$$
$$\textbf{let } d_0 : \textbf{?}[1]\textbf{N}.\textbf{End} = \textbf{send}[1](d_0, c_0) \textbf{ in}$$
$$\textbf{let } (d_0, n) : \textbf{End} \times \textbf{N} = \textbf{receive}[1](d_0) \textbf{ in}$$
$$\textbf{close}(d_0)$$

To type the second **fork**, we need to come up with a second global type that governs the communication between the third forked-off thread and the main thread:

$$G' \triangleq [0 \rightarrow 1](G \downarrow 0).[1 \rightarrow 0]\textbf{N}.\textbf{End} \qquad \text{where } G \downarrow 0 = ![1]\textbf{N}.\textbf{?}[2]\textbf{N}.\textbf{End}$$

The projections are $G' \downarrow 0 = ![1](G \downarrow 0).\textbf{?}[1]\textbf{N}.\textbf{End}$ and $G' \downarrow 1 = \textbf{?}[0](G \downarrow 0).![0]\textbf{N}.\textbf{End}$. This global type shows that participant 0 (the main thread) of G' first delegates an endpoint with local type $G \downarrow 0$ to participant 1 of G' (the third forked-off thread), which then sends a natural number back. In the code, the main thread sends endpoint c_0, which the third forked-off thread receives as x. The third forked-off thread then executes the communication according to local type $G \downarrow 0$, and sends back a natural number to the main thread.

NOVEL ELEMENTS OF MPGV   As demonstrated by the above example, MPGV's session-typed endpoints are first class and can thus be sent over channels (*i.e.*, delegated) like any other data. MPGV not only allows sending single endpoints over channels, but also lists of endpoints (Section 3.2.8) or closures, which may capture endpoints. Data types that contain endpoints are treated linearly in order to protect type safety, whereas data types that cannot contain endpoints (*e.g.*, lists of natural numbers) may be freely copied and discarded. MPGV guarantees deadlock freedom in the presence of interleaved sessions solely by linear typing and n-ary fork, and without any extrinsic mechanisms like orders/priorities (Bettini et al., 2008; Coppo et al., 2013, 2016).

3.2.4 *Participant Redirecting*

In the example from Section 3.2.2 we have two threads *service*$_1$ and *service*$_2$ that were doing more or less the same thing (adding 3 and 4, respectively). To obtain a language that enables modular programming, we would like to write a single function that generalizes both services that we could use for both threads in the **fork** operation. Let us try to make an attempt:

service : $\mathbf{N} \to (\textbf{?}[0]\mathbf{N}.\textbf{!}[1]\mathbf{N}.\text{End}) \to \mathbf{1}$

service a $c \triangleq$ **let** $(c, n) = $ **receive**$[0](c)$ **in**

         let $c = $ **send**$[1](c, n + a)$ **in**

         close(c)

The function *service* takes a natural number a for the value that should be added. Unfortunately, *service* 3 and *service* 4 cannot readily be used because their types do not match up with $G \downarrow 1 = \textbf{?}[0]\mathbf{N}.\textbf{!}[2]\mathbf{N}.\text{End}$ and $G \downarrow 2 = \textbf{?}[1]\mathbf{N}.\textbf{!}[0]\mathbf{N}.\text{End}$ since the participant numbers are off.

MPGV provides a **redirect**$[\pi](c)$ operation that allows us to locally redirect participant numbers, making it possible for a programmer to pass endpoints to destinations where different participant numbers are expected in the type signature. The informal semantics of the **redirect** operation is that any **send**$[p]$ and **receive**$[p]$ operations on $c' = $ **redirect**$[\pi](c)$ get redirected to **send**$[\pi(p)]$ and **receive**$[\pi(p)]$ on c. With MPGV's redirect operation at hand, we can change the **fork** in the first line of the example in Section 3.2.2 into:

$$\textbf{fork}(\lambda c_1.\ service\ 3\ (\textbf{redirect}[0 \mapsto 0, 1 \mapsto 2](c_1)),$$
$$\lambda c_2.\ service\ 4\ (\textbf{redirect}[0 \mapsto 1, 1 \mapsto 0](c_2)))$$

NOVEL ELEMENTS OF MPGV   Redirecting is a novel concept that has not been explored in multiparty session types to our knowledge. Redirecting is important for modularity because it allows composing a function f with a function g with compatible range and domain types even when participant numbers are at odds. Redirecting is also crucial for embedding binary sessions in MPGV; without redirecting, that would not be possible (see Section 3.4).

3.2.5 *Choice and Recursive Session Types*

Similar to traditional (multiparty) session types, MPGV supports *choice* and *recursion*. For example:

$$G'' \triangleq [0 \to 1]\{A : \mathbf{N}.G'', B : \textbf{string}.\text{End}\}$$

In this global type, participant 0 sends participant 1 a choice label $\{A, B\}$. If the choice label is A, then the payload of the message is of type $\mathbf{N}$, and the protocol recursively loops back to the initial state. If the choice label is B, then the payload of the message is of type **string**, and then the protocol ends. This gives the following local projections:

$$G'' \downarrow 0 \triangleq ![1]\{A : \mathbf{N}.(G'' \downarrow 0), B : \mathbf{string}.\mathsf{End}\}$$
$$G'' \downarrow 1 \triangleq ?[0]\{A : \mathbf{N}.(G'' \downarrow 1), B : \mathbf{string}.\mathsf{End}\}$$

With choice, not all global types one can write down are valid: all the branches of a choice must have *equal* projections for participants that are neither the sender nor the receiver of the choice. This is to ensure that each participant always has enough information to determine the type of the next message that they should send or expect to receive (Honda et al., 2008, 2016). MPGV supports recursive functions, which are crucial to provide implementations of recursive session types.

3.2.6 *Two Buyer Protocol*

The two buyer protocol is a classic example from the literature (Honda et al., 2008) with two buyers (Alice and Bob) and a Seller. The protocol has the following global type in MPGV (we use symbolic participant identifiers for readability; one can take $S = 0, A = 1, B = 2$):

$$G_{SAB} \triangleq [A \to S]\mathbf{string}.[S \to A]\mathbf{N}. [S \to B]\mathbf{N}.[A \to B]\mathbf{N}.$$
$$[B \to S]\{Yes : [S \to B]\mathbf{date}. \mathsf{End}, No : \mathsf{End}\}$$

This global protocol has the following projections for Alice, Bob, and Seller:

$$G_{SAB} \downarrow A = ![S]\mathbf{string}.?[S]\mathbf{N}.![B]\mathbf{N}.\mathsf{End}$$
$$G_{SAB} \downarrow B = ?[S]\mathbf{N}.?[A]\mathbf{N}.![S]\{Yes : ?[S]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}$$
$$G_{SAB} \downarrow S = ?[A]\mathbf{string}.![A]\mathbf{N}.![B]\mathbf{N}.?[B]\{Yes : ![B]\mathbf{date}.\mathsf{End}, No : \mathsf{End}\}$$

The participants perform the following interactions:

1. Alice tells the Seller which item she wants to buy ($[A \to S]\mathbf{string}$).

2. The Seller tells both Alice and Bob how much the item costs ($[S \to A]\mathbf{N}. [S \to B]\mathbf{N}$).

3. Alice tells Bob how much money she is willing to contribute to the purchase ($[A \to B]\mathbf{N}$).

4. Bob decides whether they can afford the item, and informs the Seller of his decision ($[B \to S]\{Yes : \ldots, No : \ldots\}$).

5. If Bob says *Yes*, the Seller sends Bob the date at which the item will be delivered and then ends the protocol ([S → B]**date**.End).

6. If Bob says *No*, the protocol ends immediately (End).

A possible implementation of the Seller is as follows:

seller : $G_{SAB} \downarrow S \to \mathbf{1}$
seller $c_S \triangleq$
   let $(c_S, \texttt{item})$: (![A]**N**.![B]**N**.?[B]{Yes : ![B]**date**.End, *No* : End}) × **string** =
      receive[A](c_S) **in**
   let c_S : ![B]**N**.?[B]{Yes : ![B]**date**.End, *No* : End}) = **send**[A]$(c_S, cost(\texttt{item}))$ **in**
   let c_S : ?[B]{Yes : ![B]**date**.End, *No* : End}) = **send**[B]$(c_S, cost(\texttt{item}))$ **in**
   match receive[B](c_S) **with** {
     ⟨Yes:c_S : ![B]**date**.End⟩ ↦ **let** c_S : End = **send**[B]$(c_S, date(\texttt{item}))$ **in close**(c_S)
     ⟨*No*:c_S : End⟩ ↦ **close**(c_S)
   }

In the case ⟨*Yes*:c_S⟩, we have c_S : ![B]**date**.End, whereas in case ⟨*No*:c_S⟩ we have c_S : End, so the type of the endpoint depends on which choice was made by Bob. Assuming that we also have functions *alice* : $G_{SAB} \downarrow A \to \mathbf{1}$ and *bob* : $G_{SAB} \downarrow B \to \mathbf{1}$ for Alice and Bob, we can run the two buyer protocol with program *seller* (**fork**(*alice, bob*)).

3.2.7 *Three Buyer Protocol and Session Delegation*

The two buyer example has been extended with delegation by Bettini et al. (2008), which means that a channel is sent as a message on another channel. To help Alice and Bob, there is a fourth person, Carol. If Bob and Alice cannot afford the item together, then instead of replying *No* to the Seller, Bob will send the remainder of his session to Carol (*i.e.,* delegation). Carol will then respond *Yes* to the Seller, if the three of them together have enough money. This is modeled by a separate session between Bob and Carol with global type:

$$G_{BC} \triangleq [B \to C](\mathbf{N} \times (![S]\{Yes : ?[S]\textbf{date}.End, \mathit{No} : End\})).End$$
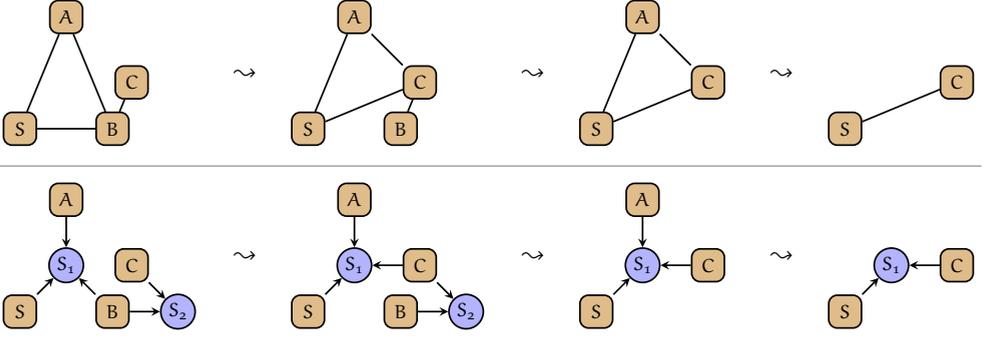
Figure 16: Steps in three buyer protocol. Top: physical communication paths; bottom: logical connectivity.

Because Bob needs access to Carol, his function is parameterized by that endpoint c_C as well as his own endpoint c_B in the two buyer protocol between him, Alice, and the Seller:

$bob_{del} : G_{BC} \downarrow B \rightarrow G_{SAB} \downarrow B \rightarrow \mathbf{1}$

$bob_{del} \; c_C \; c_B \triangleq$

    let $(c_B, \text{cost}) : (\textbf{?}[A]\textbf{N}.\textbf{!}[S]\{\text{Yes} : \textbf{?}[S]\textbf{date}.\text{End}, No : \text{End}\}) \times \textbf{N} = \textbf{receive}[S](c_B)$ **in**

    let $(c_B, \text{contrib}_A) : (\textbf{!}[S]\{\text{Yes} : \textbf{?}[S]\textbf{date}.\text{End}, No : \text{End}\}) \times \textbf{N} = \textbf{receive}[A](c_B)$ **in**

    if $\text{cost} - \text{contrib} < \text{max}_B$ **then**

      let $c_B : \textbf{?}[S]\textbf{date}.\text{End} = \textbf{send}[S](c_B, \langle \text{Yes} \rangle)$ **in**

      let $(c_B, \text{date}) : \text{End} \times \textbf{date} = \textbf{receive}[S](c_B)$ **in**

      close(c_B)

    else

      let $c_C : \text{End} = \textbf{send}[C](c_C, (\text{cost} - \text{contrib}_A - \text{max}_B, c_B))$ **in**

      close(c_C)

In the else branch, Bob sends his endpoint c_B over his connection to Carol, c_C. We can run the three buyer protocol with the following program, assuming that we have $carol : G_{BC} \downarrow C \rightarrow \mathbf{1}$:

$$\text{\textbf{let} } c_C : G_{BC} \downarrow B = \textbf{fork}(carol) \textbf{ in}$$
$$seller \; (\textbf{fork}(alice, bob_{del} \; c_C))$$

Depending on thread scheduling, operations can be executed in a different order. One possible execution is graphically depicted in the top row of Figure 16. In the left picture, we have the session between A, B, and S, and the session between B and C. In our operational semantics, the participants are connected directly, and each participant has their own set of buffers in the heap, separate from the others. At some

point Bob decides to send his session to Carol (second picture), so the connections of B get moved to C. Bob then ends his session with Carol (third picture). Alice ends her participation in the session (fourth picture). This deletes her buffers from the heap, even though the Seller and Carol may still be actively communicating. The global type ensures that whenever Alice is allowed to close her session, the other participants are guaranteed not to perform further communication with her.

3.2.8 *Endpoints in Data Structures*

Because of the functional nature of MPGV, we can freely intermix sessions and data structures. We give an example of a department store, to which we can send several buyers in a list. The department store will then let the buyers interact by applying the *seller* function for us. To illustrate recursive protocols, the department store loops around and accepts new buyers:

departmentstore : $(\mu x.\ \textbf{?}[C]List(G_{SAB} \downarrow S).x) \rightarrow \mathbf{1}$

departmentstore $c_D \triangleq \textbf{let } (c_D, \text{endpoints}) = \textbf{receive}[C](c_D) \textbf{ in}$
$\qquad\qquad\qquad \text{map } seller \text{ endpoints; } departmentstore\ c_D$

Given a function *buyers* : $\textbf{string} \rightarrow G_{SAB} \downarrow S$ that starts up the two or three buyers trying to buy an item of the given name and returns the seller's endpoint to interact with them, we can start a department store and send buyers to it as follows:

$\textbf{let } store = \textbf{fork}(departmentstore) \textbf{ in}$
$\textbf{let } c_1 = buyers \text{ "hat" in}$
$\textbf{let } c_2 = buyers \text{ "cow" in}$
$\textbf{let } store = \textbf{send}[D](store, [c_1; c_2]) \textbf{ in}$
$\textbf{let } c_3 = buyers \text{ "egg" in}$
$\textbf{let } c_3 = buyers \text{ "bow" in}$
$\textbf{let } store = \textbf{send}[D](store, [c_3; c_4]) \textbf{ in } \dots$

NOVEL ELEMENTS OF MPGV   MPGV allows *multiparty* endpoints to be stored in data structures, and captured in closures, which can then be sent as messages. This is in contrast to earlier multiparty systems, where endpoints can either not be manipulated at all (Castro-Perez et al., 2021), or where there is a separate syntactic category for endpoints, which cannot be mixed with data (Honda et al., 2008; Coppo et al., 2016; Bettini et al., 2008; Coppo et al., 2013).

3.2.9 *Deadlock Freedom of MPGV*

MPGV's deadlock freedom proof is based on two key ideas: (1) local progress *within* a session is guaranteed by the global type, and (2) global progress *between* sessions is guaranteed by our n-ary fork and linear typing, asserting that the communication topology between sessions remains acyclic (despite first-class endpoints). To reason about deadlock freedom we abstract a *logical connectivity* topology from the *physical communication* topology and prove that the logical connectivity topology remains acyclic. The logical topology of the three buyer protocol is depicted in the bottom row of Figure 16. It introduces a blue circle for each multiparty session, abstracting over the cyclic topology within a session and exposing the acyclicity of the logical topology. Figure 16 shows that the logical connectivity topology remains acyclic throughout the execution. This holds for any well-typed MPGV program—Figure 25 in Section 3.7 shows how the logical topology is transformed and remains acyclic for each of the session operations.

NOVEL ELEMENTS OF MPGV   Similar to binary variants of GV, MPGV ensures global progress and deadlock freedom for an entire program, solely by linear typing. In contrast, earlier multiparty systems either guarantee deadlock freedom only for a single session (Castro-Perez et al., 2021; Honda et al., 2008), or for multiple sessions if types are augmented with extrinsic orders/priorities (Coppo et al., 2016; Bettini et al., 2008; Coppo et al., 2013). Moreover, our global progress and deadlock freedom theorems are mechanized in Coq (Section 3.5).

3.3 THE SEMANTICS OF MPGV

3.3.1 *Syntax and Operational Semantics*

Each configuration in our small-step operational semantics consists of a *thread pool* and *heap*, which stores a vector of buffers for each endpoint:

$$\rho \in \mathsf{Cfg} \triangleq \mathsf{List}\ \mathsf{Expr} \times \mathsf{Heap}$$
$$h \in \mathsf{Heap} \triangleq \mathsf{Endpoint} \xrightarrow{\mathsf{fin}} (\mathsf{Participant} \xrightarrow{\mathsf{fin}} \mathsf{List}\ (\mathsf{Label} \times \mathsf{Val}))$$

An endpoint $c \in \mathsf{Endpoint} ::= (s, p)$ consists of a number $s \in \mathsf{Session}$ identifying the session, and a number $p \in \mathsf{Participant}$ identifying the participant number of the endpoint in the session.

The operational semantics has three reduction relations. Firstly, $e \leadsto_{\mathsf{pure}} e'$ for pure reductions of expressions. Secondly, $(e, h) \leadsto_{\mathsf{head}} (e', h', \vec{e})$ for reductions of channel operations involving the heap h, with the option to spawn a list of new threads $\vec{e}$ (a non-empty list for **fork**, and an empty list for the other operations). Thirdly, $(\vec{e}, h) \leadsto_{\mathsf{cfg}} (\vec{e}', h')$ between configurations, which performs $\leadsto_{\mathsf{head}}$ on some thread in the thread pool, and also handles evaluation contexts. The formal syntax

Expressions, values, and evaluation contexts

$$e \in \mathsf{Expr} ::= x \mid () \mid n \mid (e, e) \mid \langle \ell : e \rangle \mid \lambda x.\, e \mid \mathbf{rec}\ f\ x.\, e \mid e\ e \mid \mathbf{fork}(e, \ldots, e) \mid$$
$$\mathbf{send}[p](e, \ell : e) \mid \mathbf{receive}[p](e) \mid \mathbf{close}(e) \mid \mathbf{redirect}[\pi](e) \mid$$
$$\mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid \mathbf{match}\ e\ \mathbf{with}\ \{\langle \ell : x \rangle \mapsto e;\ \ldots\}_{\ell \in I}$$
$$v \in \mathsf{Val} ::= () \mid n \mid (v, v) \mid \langle \ell : v \rangle \mid \lambda x.\, e \mid \mathbf{rec}\ f\ x.\, e \mid \#[c, \pi]$$
$$K \in \mathsf{Ctx} ::= \square \mid (K, e) \mid (v, K) \mid K\ e \mid v\ K \mid \mathbf{let}\ x = K\ \mathbf{in}\ e \mid \cdots$$

Data structures

$$s \in \mathsf{Session} \triangleq \mathbb{N} \quad (s, p) \in \mathsf{Endpoint} \triangleq \mathsf{Session} \times \mathsf{Participant}$$
$$p, q \in \mathsf{Participant} \triangleq \mathbb{N} \quad \pi \in \mathsf{Translation} \triangleq \mathsf{Participant} \xrightarrow{\mathrm{fin}} \mathsf{Participant}$$
$$\ell \in \mathsf{Label} \triangleq \mathbb{N} \quad h \in \mathsf{Heap} \triangleq \mathsf{Endpoint} \xrightarrow{\mathrm{fin}} (\mathsf{Participant} \xrightarrow{\mathrm{fin}} \mathsf{List}\,(\mathsf{Label} \times \mathsf{Val}))$$
$$\rho \in \mathsf{Cfg} \triangleq \mathsf{List}\,\mathsf{Expr} \times \mathsf{Heap}$$

Small-step operational semantics

$$(e_1, h) \rightsquigarrow_{\mathrm{head}} (e_2, h, \epsilon) \quad (\text{if } e_1 \rightsquigarrow_{\mathrm{pure}} e_2)$$
$$(\mathbf{fork}(v_1, \ldots, v_n), h) \rightsquigarrow_{\mathrm{head}} (\#[(s, 0), \mathsf{id}], h \uplus \{(s, 0) \mapsto \vec{\epsilon}, \ldots, (s, n) \mapsto \vec{\epsilon}\},$$
$$[v_1\ \#[(s, 1), \mathsf{id}], \ldots, v_n\ \#[(s, n), \mathsf{id}]])$$
$$(\mathbf{send}[q](\#[(s, p), \pi], \ell : v), h) \rightsquigarrow_{\mathrm{head}} (\#[(s, p), \pi], \mathsf{push}((s, \pi(q)), p, \langle \ell : v \rangle, h), \epsilon)$$
$$(\mathbf{receive}[p](\#[(s, q), \pi]), h) \rightsquigarrow_{\mathrm{head}} (\langle \ell : (v, \#[(s, q), \pi]) \rangle, h', \epsilon)$$
$$(\text{if } \mathsf{pop}((s, q), \pi(p), h) = (\langle \ell : v \rangle, h'))$$
$$(\mathbf{close}(\#[(s, p), \pi]), h) \rightsquigarrow_{\mathrm{head}} ((), h \backslash \{(s, p)\}, \epsilon)$$
$$(\mathbf{redirect}[\pi_1](\#[(s, p), \pi_2]), h) \rightsquigarrow_{\mathrm{head}} (\#[(s, p), \pi_2 \circ \pi_1], h, \epsilon)$$
$$(\vec{e_a} \mathbin{+\mkern-10mu+} [K[e]] \mathbin{+\mkern-10mu+} \vec{e_b}, h) \rightsquigarrow_{\mathrm{cfg}} (\vec{e_a} \mathbin{+\mkern-10mu+} [K[e']] \mathbin{+\mkern-10mu+} \vec{e_b} \mathbin{+\mkern-10mu+} \vec{e}, h')$$
$$(\text{if } (e, h) \rightsquigarrow_{\mathrm{head}} (e', h', \vec{e}))$$

Figure 17: Syntax and operational semantics of MPGV (selected rules).

and operational semantics of MPGV can be found in Figure 17. We give an informal description of the semantics of the message-passing operations **fork**, **send**, **receive**, **close**, and **redirect** next.

FORK   The fork operation $\textbf{fork}(v_1, \ldots, , v_n)$ spawns n threads and creates a new session between the $n + 1$ endpoints. The session s has $(n + 1) \times (n + 1)$ buffers in the heap h for the $n + 1$ endpoints, such that the buffer stored at $h(s, q)(p)$ queues messages sent from p to q. Session endpoints c are represented as triples $c = \#[(s, p), \pi]$ of a session address $s \in$ Session, endpoint number $p \in$ Participant, and translation vector $\pi :$ Participant $\xrightarrow{\text{fin}}$ Participant, which is used for redirecting and initialized by **fork** to be the identity mapping. Each of the values v_i passed as arguments to **fork** must be a closure that accepts an endpoint as its argument, so that the threads run function calls $v_i \#[(s, i), \text{id}]$ for $i = 1..n$. The **fork** returns endpoint $\#[(s, 0), \text{id}]$. A usage pattern is:

$$\textbf{let } c_0 = \textbf{fork}((\lambda c_1. \ e_1), \ldots, (\lambda c_n. \ e_n)) \textbf{ in } e_0$$

SEND   The send operation $\textbf{send}[q](c, \ell : v)$ sends the message $\langle \ell : v \rangle$ to q via the endpoint $c = \#[(s, p), \pi]$ by adding the message to the end of buffer (using the operation $\text{push}((s, \pi(q)), p, \langle \ell : v \rangle, h)$ in Figure 17). The message is tagged with a label ℓ, which can influence the future actions allowed to be performed by the participant. We revisit this in detail when we introduce the typing rules. Our send operation is asynchronous. One can encode synchronous communication by inserting after each message $A \rightarrow B$ a dummy message $B \rightarrow A$ with type unit to enforce synchronization.

RECEIVE   The receive operation $\textbf{receive}[p](c)$ receives a message from p via endpoint $c = \#[(s, q), \pi]$. The receive operation takes the first message out of buffer (using the operation $\text{pop}((s, q), \pi(p), h) = (\langle \ell : v \rangle, h')$ in Figure 17). If the buffer is empty, the operation blocks until a message becomes available.

CLOSE   The close operation $\textbf{close}(c)$ deletes all the buffers from which the endpoint $c = \#[(s, q), \pi]$ receives messages, that is, it simply deletes entry $h((s, q))$ of the heap.

REDIRECT   The redirecting operation $c' = \textbf{redirect}[\pi](c)$ where $\pi \in$ Participant $\xrightarrow{\text{fin}}$ Participant redirects messages so that send and receive operations to p on c' are redirected to $\pi(p)$ on c. Operationally, it composes the translation vector of c with π:

$$\textbf{redirect}[\pi_1](\#[(s, p), \pi_2]) = \#[(s, p), \pi_2 \circ \pi_1]$$

For details, see Figure 17. This operation is required to make multiparty sessions formally subsume binary sessions (Section 3.4), but is independently useful for modular programming with first-class endpoints (Section 3.2.4), because it allows

⚙

$$\frac{\Gamma \text{ unr} \quad x \notin \Gamma}{\{x \mapsto \tau\} \cup \Gamma \vdash x : \tau} \qquad \frac{\Gamma_1 \perp \Gamma_2 \quad \Gamma_1 \vdash e_1 : \tau_1 \Rightarrow \tau_2 \quad \Gamma_2 \vdash e_2 : \tau_1 \quad (\Rightarrow) \in \{\to, \multimap\}}{\Gamma_1 \cup \Gamma_2 \vdash e_1 \; e_2 : \tau_2}$$

$$\frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2 \quad x \notin \Gamma}{\Gamma \vdash \lambda x. \; e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2 \quad \Gamma \text{ unr} \quad x \notin \Gamma}{\Gamma \vdash \lambda x. \; e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \cup \{f \mapsto (\tau_1 \to \tau_2), x \mapsto \tau_1\} \vdash e : \tau_2 \quad \Gamma \text{ unr} \quad f, x \notin \Gamma}{\Gamma \vdash \mathbf{rec} \; f \; x. \; e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau_\ell}{\Gamma \vdash \langle \ell : e \rangle : \Sigma_{\ell \in I}. \; \tau_\ell}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \quad \Gamma_1 \vdash e : \Sigma_{\ell \in I}. \; \tau_\ell \quad \forall \ell \in I. \; \Gamma_2 \cup \{x_\ell \mapsto \tau_\ell\} \vdash e_\ell : \tau' \quad x_\ell \notin \Gamma_2 \quad (I = \emptyset \to \Gamma_2 = \emptyset)}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{match} \; e \; \mathbf{with} \; \{\langle \ell : x_\ell \rangle \mapsto e_\ell; \; \ldots\}_{\ell \in I} : \tau'}$$

$$\frac{\Gamma_1 \perp \cdots \perp \Gamma_n \quad \text{consistent}(L_0, L_1, \ldots, L_n) \quad \forall p \in \{1..n\}. \; \Gamma_p \vdash e_p : L_p \multimap \mathbf{1}}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash \mathbf{fork}(e_1, \ldots, e_n) : L_0}$$

$$\frac{\Gamma \vdash e : \mathsf{End}}{\Gamma \vdash \mathbf{close}(e) : \mathbf{1}} \qquad \frac{\Gamma_1 \perp \Gamma_2 \quad \Gamma_1 \vdash e_1 : ![p]\{\ell : \tau_\ell. \; L_\ell\}_{\ell \in I} \quad \Gamma_2 \vdash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{send}[p](e_1, \ell : e_2) : L_\ell}$$

$$\frac{\Gamma \vdash e : ?[p]\{\ell : \tau_\ell. \; L_\ell\}_{\ell \in I}}{\Gamma \vdash \mathbf{receive}[p](e) : \Sigma_{\ell \in I}. \; \tau_\ell \times L_\ell} \qquad \frac{\Gamma \vdash e : \pi(L)}{\Gamma \vdash \mathbf{redirect}[\pi](e) : L}$$

Figure 18: Selected MPGV typing rules.

the programmer to pass endpoints to destinations where different endpoint numbers are expected in the type signature.

3.3.2 *Static Type System*

The functional layer of MPGV features base types, products, closures, sums, and equi-recursive types (Crary et al., 1999). The message-passing layer of MPGV features multiparty sessions with n-ary choice. Formally the types of MPGV are given by:

$$\tau \in \mathsf{Type} \underset{(\mathrm{coind})}{::=} \mathbf{1} \mid \mathbf{N} \mid \tau \times \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid \Sigma_{\ell \in I}. \; \tau_\ell \mid L \qquad ⚙$$

$$L \in \mathsf{LType} \underset{(\mathrm{coind})}{::=} ![p]\{\ell : \tau_\ell. \; L_\ell\}_{\ell \in I} \mid ?[p]\{\ell : \tau_\ell. \; L_\ell\}_{\ell \in I} \mid \mathsf{End} \qquad ⚙$$

The functional types τ and local session types L are mutually defined: functional types occur as messages in local types, and local types are functional types. To support equi-recursive types, we define Type and LType *coinductively*, allowing types

to refer to themselves (Crary et al., 1999; Gay et al., 2020; Jacobs et al., 2022b; Castro-Perez et al., 2021; Keizer et al., 2021). Mutually recursive functional types and local types can be constructed using corecursion in the meta logic (*i.e.,* Coq), so there is no explicit recursion operator. We use = to denote coinductive equivalence (*i.e.,* bisimulation). The typing rules for MPGV's judgment $\Gamma \vdash e : \tau$ are displayed in Figure 18.

Unrestricted Types

We have *linear* function types $\tau_1 \multimap \tau_2$, which must be used exactly once, and whose lambda expressions can capture linear data. We also have *unrestricted* functions $\tau_1 \to \tau_2$, which can be used any number of times (incl. zero times), but whose lambda expressions cannot capture linear data. We define the subset $\mathsf{UType} \subseteq \mathsf{Type}$ of unrestricted types as:

$$\tilde{\tau} \in \mathsf{UType} \underset{(\mathrm{coind})}{::=} \mathbf{1} \mid \mathbf{N} \mid \tilde{\tau} \times \tilde{\tau} \mid \tau \to \tau \mid \Sigma_{\ell \in I}. \tilde{\tau}_\ell \qquad \text{⚙}$$

Note that $\tau_1 \to \tau_2$ is always unrestricted, even if τ_1 and τ_2 are restricted, because closures of unrestricted function type cannot contain endpoints.

To support linear and unrestricted types in the typing judgment, context disjointness $\Gamma_1 \perp \Gamma_2$ is defined such that if Γ_1 and Γ_2 both contain variable x, the two contexts must assign equal types to x (*i.e.,* $\Gamma_1(x) = \Gamma_2(x)$), and the type they assign to x must be an unrestricted type. This ensures that the union operation $\Gamma_1 \cup \Gamma_2$ on contexts is well-defined whenever it is used in the typing rules (for instance, if $\Gamma_1 = \{x : \mathbf{N}; y : \mathbf{N}\}$ and $\Gamma_2 = \{y : \mathbf{N}\}$, then $\Gamma_1 \cup \Gamma_2 = \{x : \mathbf{N}; y : \mathbf{N}\}$). A context is unrestricted if all its types are unrestricted.

Local Types

Local types describe the protocol that an endpoint c must follow:

- If $c : ![p]\{\ell : \tau_\ell. L_\ell\}_{\ell \in I}$ then the next action on c has to be **send**$[p](c, \ell : v)$, and $v : \tau_\ell$ and the continuation type L_ℓ of c is determined by the sent label $\ell \in I$.

- If $c : ?[p]\{\ell : \tau_\ell. L_\ell\}_{\ell \in I}$ then the next action on c has to be **receive**$[p](c)$, and the received label $\ell \in I$ determines the type τ_ℓ of the value received and the next type L_ℓ of c.

- If $c :$ End then the next action on c must be **close**(c).

Due to linear typing of endpoints, we must use each endpoint variable exactly once. Like in other session typed languages, this is necessary for type safety.

⚙ ⚙

$$\frac{r \neq q \qquad \forall \ell \in I. \, G_\ell \downarrow r = L_\ell}{[r \to q]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I} \downarrow r = ![q]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}}$$

$$\frac{r \neq p \qquad \forall \ell \in I. \, G_\ell \downarrow r = L_\ell}{[p \to r]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I} \downarrow r = ?[p]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}}$$

$$\frac{r \notin \{p, q\} \quad \forall \ell \in I. \, G_\ell \downarrow r = L \quad r \text{ guards } G_\ell \quad I \neq \emptyset}{[p \to q]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I} \downarrow r = L} \qquad \frac{r \notin \text{participants}(G)}{G \downarrow r = \text{End}}$$

$$\frac{r \in \{p, q\}}{r \text{ guards } [p \to q]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I}} \qquad \frac{\forall \ell \in I. \, r \text{ guards } G_\ell}{r \text{ guards } [p \to q]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I}}$$

Figure 19: Coinductive projection (dotted line) and inductive guardedness rules (solid line).

For the typing rule of **redirect** (if $e : \pi(L)$, then **redirect**$[\pi](e) : L$), we define the action of a renaming π (not necessarily injective) on local types:

$$\pi(![p]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}) \triangleq ![\pi(p)]\{\ell : \tau_\ell. \, \pi(L_\ell)\}_{\ell \in I}$$

$$\pi(?[p]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}) \triangleq ?[\pi(p)]\{\ell : \tau_\ell. \, \pi(L_\ell)\}_{\ell \in I}$$

$$\pi(\text{End}) \triangleq \text{End}$$

Global Types and Projections

The typing rule for **fork** (Figure 18) requires a session's local types $L_0, \ldots, L_n$ to be *consistent*. Consistency means, for instance, that if participant p sends a value of type τ to participant q, then q is expecting to receive a value of type τ from p at that point in the protocol. Traditionally, consistency is defined by the existence of a global type that governs the communication between all participants in a session. Global types are of the form:

$$G \in \text{GType} \underset{(\text{coind})}{::=} [p_1 \to p_2]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I} \mid \text{End} \qquad \qquad ⚙$$

A global type $[p_1 \to p_2]\{\ell : \tau_\ell. \, G_\ell\}_{\ell \in I}$ expresses that the first action in the protocol is for participant p_1 to send a message to p_2, such that if the label in the message is chosen to be ℓ, then the payload of the message has to have type τ_ℓ, and then the global protocol continues as G_ℓ. Note that "global" in our use of "global type" means global with respect to a session, not the whole program—each different session started by a **fork** can have its own global type.

Local types can be extracted from global types by a *projection judgment* $G \downarrow p = L$, indicating that participant p's local type is L if the global type is G. The judgment is coinductively defined in Figure 19. The first two rules state how the sender and

receiver of a message in the global type are projected. The third rule states how other participants not involved in the message are projected. For participants not involved in the message, we require that participant to *guard* the rest of the global type, which means that the participant occurs in the global type at finite depth along every branch. The fourth rule states that if a participant does not occur in the global type, then it projects to End. Our projection rules are similar to those of Zooid (Castro-Perez et al., 2021).

Traditionally, consistency consistent$(L_0, \ldots, L_n)$ is expressed in terms of a global type G such that $G \downarrow 0 = L_0, \ldots, G \downarrow n = L_n$, and $G \downarrow m = \text{End}$ for $m > n$. In Section 3.6 we develop, inspired by Scalas and Yoshida (2019), a more permissive notion of consistency that is independent of a global type, permitting deadlock-free scenarios for which no appropriate global type can be found. Section 3.6.2 then shows that the traditional notion of consistency based on global types implies our new notion.

3.4 TRANSLATION FROM BINARY TO MULTIPARTY

We show that a GV-style *binary* session-typed language falls out as a special mode of use of our *multiparty* language MPGV by giving a *type-preserving translation* of binary channel operations into MPGV. We consider this an important benchmark, because traditional multiparty systems do not support such a translation in the deadlock-free setting. There are two main obstacles in existing systems: (1) after translation, participant numbers do not match up, and (2) in systems such as Coppo et al. (2013); Bettini et al. (2008); Coppo et al. (2016), deadlock-freedom mechanisms such as orders/priorities prevent programs from being translated because these orders are absent in the source program, so after translation one must come up with an order on sessions. The latter is not always possible if sessions are used in a different orders in different branches of a conditional. Finally, translation of an expressive language such as GV requires the target language to support storing endpoints in data structures, as GV supports this. MPGV overcomes all these obstacles.

We start by giving a short introduction to binary session types, and then show how they can be translated into our language, making use of **redirect**. Binary session types are equivalent to local types *without* participant annotations. The annotations are not necessary in the binary case, because there is only one other participant to communicate with:

$$B \in \text{BType} \underset{(\text{coind})}{::=} \ !\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I} \mid \ ?\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I} \mid \text{End} \qquad \var�$$

✿ ✿ ✿ ✿

$$\frac{\Gamma \vdash e : [\![\overline{B}]\!]_L \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}_B(e) : [\![B]\!]_L} \qquad \frac{\Gamma \vdash e : [\![End]\!]_L}{\Gamma \vdash \mathbf{close}_B(e) : \mathbf{1}}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : [\![!\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_L \quad \Gamma_2 \vdash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vdash \mathbf{send}_B(e_1, \ell : e_2) : [\![B_\ell]\!]_L} \qquad \frac{\Gamma \vdash e : [\![?\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_L}{\Gamma \vdash \mathbf{receive}_B(e) : \Sigma_{\ell \in I}.\, \tau_\ell \times [\![B_\ell]\!]_L}$$

Figure 20: Derivable typing rules for binary session types.

The operations for binary channels are defined in terms of multiparty operations as follows:

$$\mathbf{fork}_B(e) \triangleq \mathbf{redirect}[1 \mapsto 0](\mathbf{fork}(e)) \qquad ✿$$
$$\mathbf{send}_B(e_1, \ell : e_2) \triangleq \mathbf{send}[0](e_1, \ell : e_2) \qquad ✿$$
$$\mathbf{close}_B(e) \triangleq \mathbf{close}(e) \qquad ✿$$
$$\mathbf{receive}_B(e) \triangleq \mathbf{receive}[0](e) \qquad ✿$$

We do a binary spawn using the n-ary fork, then the local type of the endpoint of the spawner gets annotated with 1's (because it is communicating with endpoint 1) and the local type of the endpoint of the forked-off thread gets annotated with 0's (because it is communicating with endpoint 0). In order to implement a type-preserving translation, we redirect all annotations to 0. This enables us to canonically translate binary session types B to multiparty local types $[\![B]\!]_L$ by using $p = 0$ for every participant annotation:

$$[\![!\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_L \triangleq !\,[0]\{\ell : \tau_\ell.\, [\![L_\ell]\!]_L\}_{\ell \in I} \qquad ✿$$
$$[\![?\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_L \triangleq ?\,[0]\{\ell : \tau_\ell.\, [\![L_\ell]\!]_L\}_{\ell \in I}$$
$$[\![End]\!]_L \triangleq End$$

We then prove that the usual typing rules for binary session types are derivable in our system. For **fork**, this amounts to defining a global type $[\![B]\!]_G$ to govern the binary interaction:

$$[\![!\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_G \triangleq [0 \rightarrow 1]\{\ell : \tau_\ell.\, [\![B_\ell]\!]_G\}_{\ell \in I} \qquad ✿$$
$$[\![?\{\ell : \tau_\ell.\, B_\ell\}_{\ell \in I}]\!]_G \triangleq [1 \rightarrow 0]\{\ell : \tau_\ell.\, [\![B_\ell]\!]_G\}_{\ell \in I}$$
$$[\![End]\!]_G \triangleq End$$

After redirecting, the projections have the right local types for B and the dual $\overline{B}$ (flips all ? with ! and vice versa):

Lemma 3.4.1. ✿ ✿ $[\![B]\!]_G \downarrow 0 = \pi^{-1}([\![B]\!]_L)$ *and* $[\![B]\!]_G \downarrow 1 = [\![\overline{B}]\!]_L$

Using this lemma and translation of types, we can prove that the binary typing rules for **fork**$_B$, **send**$_B$, **receive**$_B$ and **close**$_B$ are derivable (Figure 20).

This section shows that MPGV supports the full power of GV-style binary session types, including treatment of sessions as first-class data and dynamic spawning of sessions. Note that redirecting is crucial: without it we are not able to do a type-preserving translation, because local types **!**[0] and **?**[0] are incompatible with **!**[1] and **?**[1].

3.5 THE DEADLOCK AND LEAK FREEDOM THEOREM

MPGV guarantees strong properties for well-typed programs, while supporting dynamic spawning, session interleaving, and first-class endpoints. These properties are:

TYPE SAFETY: The only way for a thread to get stuck is by blocking to receive from an empty buffer.

SESSION FIDELITY: The values sent to and received from buffers match the types in the protocol.

GLOBAL PROGRESS: Configurations of a well-typed initial program are either final or can take a step.

DEADLOCK FREEDOM: No subset of the threads get stuck by waiting for each other.

MEMORY LEAK FREEDOM: All data always remains reachable.

Ideally, we would like to capture these properties in a single theorem that subsumes them all. As a first step, we formulate global progress as follows:

Theorem 3.5.1 (Global progress ⚙). *If* $\emptyset \vdash e : \mathbf{1}$, *and* $([e], \emptyset) \leadsto^*_{\mathrm{cfg}} (\vec{e}, h)$, *then:*

1. *there exists* $(\vec{e}', h')$ *such that* $(\vec{e}, h) \leadsto_{\mathrm{cfg}} (\vec{e}', h')$, *or*

2. $\vec{e}_i = ()$ *for all* $i \in \mathrm{dom}(\vec{e})$ *and* $h = \emptyset$.

This theorem rules out whole-program deadlocks and ensures that all buffers have been correctly deallocated when the program finishes. However, this theorem does not guarantee anything as long as there is still a single thread that can step. Thus it does not guarantee local deadlock freedom, nor memory leak freedom while the program is still running. Moreover, it does not even guarantee type safety: a situation in which a thread is stuck on a type error is not ruled out by this theorem as long as there is another thread that can still step. We therefore state partial deadlock freedom and memory leak freedom theorems, but we strengthen both so that they become equivalent. We use the definitions of partial deadlock and memory leak freedom of Jacobs et al. (2022b) and apply them to MPGV. We need the following notions:

- The set $v \in V ::= \text{Thread}(i) \mid \text{Session}(s)$ ranging over possible threads and sessions.

- The function $\text{refs}_{(\vec{e},h)}(v) \subseteq V$ giving the set of sessions that v references.

- The predicate $\text{blocked}_{(\vec{e},h)}(v_1, v_2)$ stating that thread $v_1 = \text{Thread}(i)$ is blocked on session $v_2 = \text{Session}(s)$.

- The function $\text{active}(\vec{e}, h) \subseteq V$ giving the set of active threads and sessions in the configuration.

Using these notions, we strengthen partial deadlock freedom to incorporate aspects of memory leak freedom.

Definition 3.5.2 (Partial deadlock/leak ✿). Given a configuration $(\vec{e}, h)$, a subset $S \subseteq V$ of the threads and sessions is in a partial *deadlock/leak* if the following conditions hold:

1. We have $\emptyset \subset S \subseteq \text{active}(\vec{e}, h)$.

2. For all threads $\text{Thread}(i) \in S$, the expression e_i cannot step in the heap h.

3. If $\text{Thread}(i) \in S$ and $\text{blocked}_{(\vec{e},h)}(\text{Thread}(i), \text{Session}(s))$, then $\text{Session}(s) \in S$.

4. If $\text{Session}(s) \in S$ and $\text{Session}(s) \in \text{refs}_{(\vec{e},h)}(v)$, then $v \in S$.

Definition 3.5.3 (Partial deadlock/leak freedom ✿). A configuration $(\vec{e}, h)$ is *deadlock/leak free* if no $S \subseteq V$ is in a partial deadlock/leak in $(\vec{e}, h)$.

Conversely, we strengthen memory leak freedom (*i.e.,* full reachability) to incorporate aspects of deadlock freedom.

Definition 3.5.4 (Reachability ✿). We inductively define the threads and sessions *reachable* in $(\vec{e}, h)$:

1. $\text{Thread}(i)$ is reachable if either
   - the expression e_i can step in the heap h, or
   - there exists an s such that $\text{Session}(s)$ is reachable and $\text{blocked}_{(\vec{e},h)}(\text{Thread}(i), \text{Session}(s))$ holds.

2. $\text{Session}(s)$ is reachable if there exists a reachable v such that $\text{Session}(s) \in \text{refs}_{(\vec{e},h)}(v)$.

Definition 3.5.5 (Full reachability ✿). A configuration $(\vec{e}, h)$ is *fully reachable* if all $v \in \text{active}(\vec{e}, h)$ are reachable in $(\vec{e}, h)$.

As in Jacobs et al. (2022b)'s language for binary sessions, the strengthened versions of deadlock freedom and full reachability are equivalent, and well-typed MPGV programs satisfy both properties:

Theorem 3.5.6. ⚙

A configuration $(\vec{e}, h)$ is deadlock/leak free if and only if it is fully reachable.

Theorem 3.5.7. ⚙

If $\emptyset \vdash e : \mathbf{1}$ and $([e], \emptyset) \leadsto_{cfg} (\vec{e}, h)$, then $(\vec{e}, h)$ is fully reachable and deadlock/leak free.

The final theorem encompasses type safety, session fidelity, deadlock freedom, and memory leak freedom. Global progress (Theorem 3.5.1) also follows as a corollary from the final theorem.

3.6 EXTENSION: CONSISTENCY WITHOUT GLOBAL TYPES

Inspired by Scalas and Yoshida (2019), we define a notion of consistency that does not rely on global types. This notion of consistency plays an important role in our proof of deadlock freedom (Section 3.7), but is also more flexible. It is more flexible in the sense that consistent($L_0, \ldots, L_n$) (premise of **fork** in Figure 18) may hold even if no global type exists whose projections are $L_1, \ldots, L_n$. For example, there exists no global type for the local types $L_0 = ![1]\mathbf{N}.?[1]\mathbf{N}.\mathsf{End}$ and $L_1 = ![o]\mathbf{N}.?[o]\mathbf{N}.\mathsf{End}$ because they both start with a send. Nevertheless, it would be safe and deadlock free to allow this protocol, given an asynchronous semantics.[2] The more flexible notion of consistency we define in Section 3.6.1 does allow this protocol. In Section 3.6.2 we show that the existence of a global type for local types implies our flexible notion of consistency.

3.6.1 *Defining Consistency without Global Types*

At a high level, we define consistent($L_0, \ldots, L_n$) as follows:

> "The local types $L_0, \ldots, L_n$ of a session are consistent if no deadlock can occur *within* the session when considering all possible interleavings of participant actions, assuming that each participant p follows its respective local type L_p."

Our goal is to define this notion solely as a property of the local types $L_0, \ldots, L_n$, so that consistency of a session's local types can be proven without considering other sessions. To do so, we define the notion of *shadow buffers*:

$$\hat{Q} \in \mathsf{ShadowBuf} \triangleq \mathsf{Participant} \xrightarrow{\mathsf{fin}} (\mathsf{Participant} \xrightarrow{\mathsf{fin}} \mathsf{List}(\mathsf{Label} \times \mathsf{Type}))$$

Shadow buffers are similar to the physical buffers in the heap, but there are two differences. First, whereas the physical buffers contain pairs $\langle \ell : v \rangle$ of labels and values, shadow buffers contain pairs $\langle \ell : \tau \rangle$ of labels and types. Second, whereas the

---

2 There exist other extensions of (multiparty) session types that allow for a more flexible notion of consistency. In particular, session-type systems with *asynchronous subtyping* also support this example (Ghilezan et al., 2021; Mostrous et al., 2009).

heap concerns all sessions, shadow buffers only concern a single session. Hence, the heap ranges over endpoints (recall that Endpoint ≜ Session × Participant), but shadow buffers range over mere participants.

Shadow buffers allow us to simulate the local execution of a session on the abstract level. If all the possible local executions allowed by a set of local types $\mathbf{L}$: Participant $\xrightarrow{\text{fin}}$ LType on a set of shadow buffers $\hat{Q}$ are type safe and deadlock free, we say that $\hat{Q}$ is consistent with $\mathbf{L}$, which we denote by consistent($\hat{Q}, \mathbf{L}$), and define as follows:

Definition 3.6.1. ⚙ The judgment consistent($\hat{Q}, \mathbf{L}$) is defined as the most permissive relation satisfying the following properties:

1. Consistency is preserved by sends, *i.e.,* for every participant p with $\mathbf{L}(p) = ![q]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}$, then consistent(push(q, p, $\langle \ell : \tau_\ell \rangle$, $\hat{Q}$), $\mathbf{L}[p := L_\ell]$).

2. Consistency is preserved by receives, *i.e.,* for every participant q with $\mathbf{L}(q) = ?[p]\{\ell : \tau_\ell. \, L_\ell\}_{\ell \in I}$, and pop(q, p, $\hat{Q}$) = ($\langle \ell : \tau \rangle$, $\hat{Q}'$), then $\ell \in I$, and consistent($\hat{Q}', \mathbf{L}[q := L_\ell]$), and $\tau = \tau_\ell$.

3. Consistency is preserved by channel closure, *i.e.,* for every participant p with $\mathbf{L}(p) = \text{End}$, then consistent($\hat{Q} \backslash \{p\}, \mathbf{L} \backslash \{p\}$).

4. Either all buffers have been deallocated ($\hat{Q} = \emptyset$) or there is a participant q such that q's local type $\mathbf{L}(q)$ is a send or a close, or $\mathbf{L}(q)$ is a receive and the corresponding buffer contains a value, *i.e.,* pop(q, p, $\hat{Q}$) = ($\langle \ell : \tau \rangle$, $\hat{Q}'$) for some label ℓ, type τ, and new set of shadow buffers $\hat{Q}'$.

5. For each participant there is a corresponding set of buffers and vice versa, *i.e.,* dom($\mathbf{L}$) = dom($\hat{Q}$).

Note that the cases for the preservation of consistent($\hat{Q}, \mathbf{L}$) under the sends, receives, and channel closure refer to a recursive occurrence consistent($\hat{Q}', \mathbf{L}'$) for some $\hat{Q}'$ and $\mathbf{L}'$. Since we consider the most permissive relation, these recursive occurrences should be interpreted coinductively—we use Coq's `CoInductive` keyword in the mechanization.

The first three properties are used to show that the channel operations are type safe and the resulting state is again consistent. The fourth property is used to show deadlock freedom. The fifth property is required for technical reasons because we support the possibility of some participants deallocating their buffers while other participants are continuing to communicate with each other. With this at hand, we define the new consistency predicate used in the **fork** typing rule:

Definition 3.6.2. ⚙ We define consistent($\vec{L}$) as consistent(init(length($\vec{L}$)), $\vec{L}$), where init(n) creates n empty buffers, and the list $\vec{L}$ is converted into a map in the natural way.

Note that we need to use the finite map representation because some participants can close their channel before others (see Item 3 in Theorem 3.6.1), and then they disappear from $\mathbf{L}$ (lists do not allow gaps in the middle, whereas finite maps do).

3.6.2 *Global Types Imply Consistency*

The goal of this section is to show that if there is a global type for a set of local types, then the local types are consistent in the sense of the preceding section:

Theorem 3.6.3. ✿ *If there is a global type* G *with* $n + 1$ *participants such that* $G \downarrow 0 = L_0, \ldots, G \downarrow n = L_n$, *then* consistent$(L_0, \ldots, L_n)$.

This lemma shows that we did not lose anything by using the more flexible notion of consistency without global types—the programs we are able to type check with the more flexible notion of consistency are a superset of the programs we are able to type check using global types.

We cannot prove Theorem 3.6.3 directly using coinduction, because the coinductive conclusion is not general enough. We need a more general property that involves the consistency judgment consistent$(\hat{Q}, L)$ for an arbitrary set of shadow buffers $\hat{Q}$. Our generalized property (Theorem 3.6.4) makes use of the notion *runtime global types*, inspired by the work of Castro-Perez et al. (2021).

RUNTIME GLOBAL TYPES To model the state of a global type during an interaction in which some messages have already been sent but not yet received, we define runtime global types as:

$$R \in \text{RType} \underset{\text{(ind)}}{::=} [p_1 \xrightarrow{\ell?} p_2]\{\ell : \tau_\ell. R_\ell\}_{\ell \in I} \mid \text{Cont } G \qquad \text{✿}$$

Runtime global types differ from ordinary global types (Section 3.3.2) in three aspects:

1. Operations in runtime global types have an optional label ℓ on the arrow. If no label is present (*i.e.,* $[p_1 \to p_2]\{\ell : \tau_\ell. R_\ell\}_{\ell \in I}$), then both the send and receive remain to happen. If a label ℓ is present (*i.e.,* $[p_1 \xrightarrow{\ell} p_2]\{\ell : \tau_\ell. R_\ell\}_{\ell \in I}$), then the send portion (with label ℓ) of the operation has already happened, but the receive is still pending.

2. Runtime global types are defined *inductively* rather than coinductively, because only finitely many messages have been sent at any given point in time.

3. Instead of having End, they have Cont G, indicating that the protocol continues as ordinary global type G.

RUNTIME LOCAL TYPE PROJECTIONS ✿

The projections $R \downarrow p = L$ of runtime global types onto local types can be found in Figure 21. These rules are inductively defined. Intuitively, when an operation $[p \xrightarrow{\ell} q]\{\ell : \tau_\ell. R_\ell\}_{\ell \in I} \downarrow r$ occurs in the runtime global type, then the projection onto p ignores the operation and continues with R_ℓ because the send by p with label ℓ has

❖ ❖

$$\frac{q \neq r \qquad \forall \ell \in I.\, R_\ell \downarrow r = L_\ell}{[r \to q]\{\ell : \tau_\ell.\, R_\ell\} \downarrow r = ![q]\{\ell : \tau_\ell.\, L_\ell\}} \qquad \frac{p \neq r \qquad \forall \ell \in I.\, R_\ell \downarrow r = L_\ell}{[p \xrightarrow{\ell?} r]\{\ell : \tau_\ell.\, R_\ell\} \downarrow r = ?[p]\{\ell : \tau_\ell.\, L_\ell\}}$$

$$\frac{r \notin \{p, q\} \qquad \forall \ell \in I.\, R_\ell \downarrow r = L \qquad I \neq \emptyset}{[p \to q]\{\ell : \tau_\ell.\, R_\ell\} \downarrow r = L} \qquad \frac{q \neq r \qquad R_\ell \downarrow r = L}{[p \xrightarrow{\ell} q]\{\ell : \tau_\ell.\, R_\ell\} \downarrow r = L}$$

$$\frac{G \downarrow r = L}{\text{Cont } G \downarrow r = L} \qquad \frac{\text{pop}(q, p, \hat{Q}) = \bot \qquad \forall \ell.\, R_\ell \Downarrow \hat{Q}}{[p \to q]\{\ell : \tau_\ell.\, R_\ell\} \Downarrow \hat{Q}}$$

$$\frac{\text{pop}(q, p, \hat{Q}) = (\langle \ell : \tau_\ell \rangle, \hat{Q}') \qquad R_\ell \Downarrow \hat{Q}'}{[p \xrightarrow{\ell} q]\{\ell : \tau_\ell.\, R_\ell\} \Downarrow \hat{Q}} \qquad \frac{\hat{Q} = \emptyset}{\text{Cont } G \Downarrow \hat{Q}}$$

Figure 21: Projections of runtime global types: (1) local type projections $R \downarrow p = L$, and (2) shadow buffer projections $R \Downarrow \hat{Q}$ (inductive).

already happened. However, the projection onto q in this case still has to take the receive part of this operation into account, because the receive has not happened yet. The other cases are similar to the projections for ordinary global types (Figure 18), and ensure that the protocol remains well-formed.

RUNTIME BUFFER PROJECTIONS ❖

We also define the judgment $R \Downarrow \hat{Q}$, which says that the messages in the runtime global type R correspond to the shadow buffers $\hat{Q}$.

RUNTIME GLOBAL TYPES IMPLY CONSISTENCY   Using the notion of runtime global type and runtime projections, we are able to show the following lemma:

Lemma 3.6.4. ❖ *The judgment* consistent($\hat{Q}$, **L**) *holds if there exists a runtime global type R for which the following four conditions hold: (1)* $R \Downarrow \hat{Q}$ *(2)* $\forall p.\, R \downarrow p = \mathbf{L}(p)$ *(3)* participants(R) $\subseteq$ dom(**L**) *(4)* $\forall p.\, if\, \hat{Q}(p) = \bot\, then\, p \notin$ dom(**L**) *else* dom(**L**) $\subseteq$ dom($\hat{Q}(p)$).

The lemma is proved using coinduction, and relies on a series of auxiliary lemmas. Once we have this lemma, Theorem 3.6.3 follows by relating projection of runtime global types to projection of ordinary global types.

3.7 PROOF OF DEADLOCK AND LEAK FREEDOM

We give an overview of the proof of our main result, Theorem 3.5.7. The proof is quite technical, but since all parts have been mechanized in Coq, one can trust the theorems independent of the pen-and-paper description of the proof. We hope to provide enough insights into the proof to make our results reproducible and extensible.

The high level structure of the proof is as follows:

- We define an *invariant* on the runtime configuration, which states (1) that everything in the configuration is well-typed and that the buffer contents are consistent with respect to the local types of each channel endpoint, and (2) that the topology of the configuration is acyclic.

- We prove that the invariant is preserved by steps of the operational semantics ("preservation").

- We prove that configurations that satisfy the invariant cannot be in a deadlock ("progress").

To deal with linearity and acyclicity we use the connectivity graph framework of Jacobs et al. (2022b), which provides a couple of features to make our proof feasible. First, it provides a generic construction to define the invariant—it allows us to provide local invariants for threads and channels, which the framework then lifts to an invariant for whole runtime configurations. Second, it makes use of separation logic to hide reasoning about linearity. Third, it provides generic reasoning principles to prove the preservation (of acyclicity and typing) and progress parts of the proof. Fourth, it is implemented as a library in Coq, so it allows us to mechanize our proofs.

At the high-level, the structure of our proof and our use of the connectivity framework is similar to Jacobs et al. (2022b)'s proof for binary session types. To use the framework to obtain the invariant for configurations (Section 3.7.3), we first define a *runtime type system* for expressions to express the local invariant for threads (Section 3.7.1), and define a local invariant for the buffers that back a session (Section 3.7.2). The new element of our proof is handling multiparty instead of binary sessions, for which we make use of our notion of shadow buffers (Section 3.6).

With the invariant for configurations at hand, we prove that this invariant holds for the initial configurations and is preserved by the operational semantics (Section 3.7.4). The new element is an extension of the connectivity graph framework to handle n-ary graph transformations to support the multiparty case. To complete the proof, we show that the configuration invariant implies Theorem 3.5.7, our main deadlock freedom theorem (Section 3.7.5).

$$P, Q \in iProp \triangleq (V \xrightarrow{\text{fin}} E) \to Prop \qquad V ::= Thread(i) \mid Session(s)$$

$$(Emp)(\Sigma) \triangleq (\Sigma = \emptyset) \qquad\qquad E \triangleq Participant \times LType$$

$$(False)(\Sigma) \triangleq False \qquad\qquad \Sigma \in V \xrightarrow{\text{fin}} E$$

$$(True)(\Sigma) \triangleq True \qquad\qquad (P \vee Q)(\Sigma) \triangleq P(\Sigma) \vee Q(\Sigma)$$

$$(\ulcorner \phi \urcorner)(\Sigma) \triangleq \phi \wedge (\Sigma = \emptyset) \qquad (P \wedge Q)(\Sigma) \triangleq P(\Sigma) \wedge Q(\Sigma)$$

$$(own(\Sigma'))(\Sigma) \triangleq (\Sigma = \Sigma') \qquad (\exists x.\, P(x))(\Sigma) \triangleq \exists x.\, P(x)(\Sigma)$$

$$(\Box P)(\Sigma) \triangleq P(\emptyset) \wedge \Sigma = \emptyset \qquad (\forall x.\, P(x))(\Sigma) \triangleq \forall x.\, P(x)(\Sigma)$$

$$(P * Q)(\Sigma) \triangleq \exists \Sigma_1 \Sigma_2.\, dom(\Sigma_1) \cap dom(\Sigma_2) = \emptyset \wedge \Sigma = \Sigma_1 \uplus \Sigma_2 \wedge P(\Sigma_1) \wedge Q(\Sigma_2)$$

$$(P \twoheadrightarrow Q)(\Sigma) \triangleq \forall \Sigma'.\, \big(dom(\Sigma) \cap dom(\Sigma') = \emptyset \wedge P(\Sigma')\big) \Rightarrow Q(\Sigma \uplus \Sigma')$$

Figure 22: The definition of the separation logic connectives.

3.7.1 *Runtime Type System*

The first step to define the invariant for configurations is to define a runtime typing judgment for expressions. The runtime judgment differs from the static typing judgment (Section 3.3.2) in the sense that it should account for channel literals #[c, π] that appear after the execution of a **fork**. Traditionally, this is done by extending the typing judgment $\Sigma; \Gamma \vdash e : \tau$ with an additional context Σ that keeps track of the types of the channel literals (often called a *heap typing*).[3] To avoid having to thread through such this context everywhere, and having to deal with splitting conditions of this context (due to linearity), we make use of separation logic (O'Hearn and Pym, 1999; O'Hearn et al., 2001). This follows the approach in the connectivity graph framework (Jacobs et al., 2022b), which in turn is based on Rouvoet et al. (2020)'s use of separation logic to hide heap typings in intrinsically-typed interpreters for linear languages in Agda.

Our runtime judgment $\Gamma \vDash e : \tau$ is formalized as a separation logic proposition iProp, *i.e.*, a predicate over heap typings Σ. The semantics of the separation logic connectives can be found in Figure 22 and the rules of our runtime type system in Figure 23. Crucially, the use of separating conjunction in the rules of n-ary constructs hides the splitting of the heap typing Σ, and the use of $own(s \mapsto (p, \pi(L)))$ in the rule for endpoint literals #[(s, p), π] makes sure the type of each literal matches up with the heap typing Σ. Note that the runtime judgment $\Gamma \vDash e : \tau$ is defined recursively on the structure of e. To assert that $P \in iProp$ is true, means to assert that $P(\emptyset)$ holds.

To prove the initialization lemma (Theorem 3.7.4), we state in separation logic that statically well-typed expressions are well-typed in the runtime type system:

---

3 The actual type of Σ in Figure 22 also accounts for threads in addition to sessions. This is due to the use of the connectivity graph framework, which we discuss in Section 3.7.3.

⚙

$$\frac{\ulcorner \Gamma \text{ unr} \urcorner \quad * \quad \text{own}(s \mapsto (p, \pi(L)))}{\Gamma \vDash \#[(s, p), \pi] : L} *$$

$$\frac{\Box(\Gamma \cup \{x \mapsto \tau_1\} \vDash e : \tau_2) \quad * \quad \ulcorner \Gamma \text{ unr} \urcorner \quad * \quad \ulcorner x \notin \Gamma \urcorner}{\Gamma \vDash \lambda x.\, e : \tau_1 \to \tau_2} *$$

$$\frac{\ulcorner \Gamma_1 \perp \cdots \perp \Gamma_n \urcorner * \ulcorner \text{consistent}(L_0, L_1, \ldots, L_n) \urcorner * [*]\, p \in \{1..n\}.\ \Gamma_p \vDash e_p : L_p \multimap \mathbf{1}}{\Gamma_1 \cup \cdots \cup \Gamma_n \vDash \mathbf{fork}(e_1, \ldots, e_n) : L_0} *$$

$$\frac{\Gamma \vDash e : \text{End}}{\Gamma \vDash \mathbf{close}(e) : \mathbf{1}} * \qquad \frac{\ulcorner \Gamma_1 \perp \Gamma_2 \urcorner \quad * \quad \Gamma_1 \vDash e_1 : ![p]\{\ell : \tau_\ell.\, L_\ell\}_{\ell \in I} \quad * \quad \Gamma_2 \vDash e_2 : \tau_\ell}{\Gamma_1 \cup \Gamma_2 \vDash \mathbf{send}[p](e_1, \ell : e_2) : L_\ell} *$$

$$\frac{\Gamma \vDash e : ?[p]\{\ell : \tau_\ell.\, L_\ell\}_{\ell \in I}}{\Gamma \vDash \mathbf{receive}[p](e) : \Sigma_{\ell \in I}.\, \tau_\ell \times L_\ell} * \qquad \frac{\Gamma \vDash e : \pi(L)}{\Gamma \vDash \mathbf{redirect}[\pi](e) : L} *$$

Figure 23: Selected separation logic runtime typing rules (recursive).

Lemma 3.7.1. ⚙ $\ulcorner \Gamma \vdash e : \tau \urcorner \twoheadrightarrow \Gamma \vDash e : \tau$

3.7.2 *The Buffer Invariant*

We now define an invariant consistent$(Q, \mathbf{L})$ to express that the buffers Q for a given session s are consistent with respect to a set of local types $\mathbf{L} : \text{Participant} \xrightarrow{\text{fin}} \text{LType}$. The buffer invariant is similar to the consistency judgment consistent$(\hat{Q}, \mathbf{L})$ we defined in Section 3.6.1, but it operates on physical buffers Q (*i.e.,* buffers with values) instead of shadow buffers $\hat{Q}$ (*i.e.,* buffers with types):

$$Q \in \text{Buf} \triangleq \text{Participant} \xrightarrow{\text{fin}} (\text{Participant} \xrightarrow{\text{fin}} \text{List}(\text{Label} \times \text{Val}))$$

(We use the notation $h|_s$ to obtain the buffers for a session s from the heap h.)

Since MPGV allows to send arbitrary data over channels, the values in buffers can themselves contain channel literals. Hence, similar to the runtime typing judgment, the buffer invariant needs to be indexed by a heap typing Σ, which we hide again by considering consistent$(Q, \mathbf{L})$ to be a separation logic proposition iProp. The definition of consistent$(Q, \mathbf{L}) \in$ iProp can be found in Figure 24. This definition contains two key ingredients. First, it makes use of consistent$(\hat{Q}, \mathbf{L}) \in$ Prop from Section 3.6 to specify that local types $\mathbf{L}$ are consistent with some (existentially quantified) shadow buffers $\hat{Q}$. Second, it makes use of the auxiliary definition $Q \propto \hat{Q} \in$ iProp in Figure 24 to specify that the labels in the physical buffers Q are equal to those in the shadow buffers $\hat{Q}$, and that the values in the physical buffers Q have types determined by the corresponding entry in the shadow buffers $\hat{Q}$. (The notation $[*]\, x; y \in X; Y.\ P(x, y)$ in

$$\mathsf{wf}(\vec{e}, h) \triangleq \mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e},h)})$$

$$\mathsf{wf}(P) \triangleq \exists G : \mathsf{Cgraph}(V, E). \forall v \in V. P(v, \mathsf{in}(G, v))(\mathsf{out}(G, v))$$

$$\mathsf{wf}^{local}_{(\vec{e},h)}(v, \Delta) \triangleq \begin{cases} \ulcorner\Delta = \emptyset\urcorner * (\emptyset \vDash e_i : \mathbf{1}) & \text{if } v = \mathsf{Thread}(i), i < |\vec{e}| \\ \ulcorner\Delta = \emptyset\urcorner & \text{if } v = \mathsf{Thread}(i), i \geqslant |\vec{e}| \\ \exists L \in \mathsf{Participant} \xrightarrow{\mathsf{fin}} \mathsf{LType}. & \text{if } v = \mathsf{Session}(s) \\ \quad \ulcorner\Delta = \mathsf{toMultiset}(L)\urcorner * \mathsf{consistent}(h|_s, L) \end{cases}$$

$$\mathsf{consistent}(Q, L) \triangleq \exists \hat{Q}. \ulcorner\mathsf{consistent}(\hat{Q}, L)\urcorner * Q \propto \hat{Q}$$

$$Q \propto \hat{Q} \triangleq [*] Q_p; \hat{Q}_p \in Q; \hat{Q}. [*] Q_{pq}; \hat{Q}_{pq} \in Q_p; \hat{Q}_p.$$
$$[*] \langle \ell_1 : v \rangle; \langle \ell_2 : \tau \rangle \in Q_{pq}; \hat{Q}_{pq}. \ulcorner\ell_1 = \ell_2\urcorner * (\emptyset \vDash v : \tau)$$

Figure 24: Configuration invariant.

Figure 24 is an n-ary separating conjunction: it states that the collections X, Y (lists or finite maps) have the same domain, and gives $P(X_0, Y_0) * \cdots * P(X_n, Y_n)$, where (X_i, Y_i) are corresponding elements in the collections.)

The invariant $\mathsf{consistent}(Q, L)$ for physical buffers has preservation and initialization properties paralleling to the rules of the consistency relation $\mathsf{consistent}(\hat{Q}, L)$ for shadow buffers (Theorems 3.6.1 and 3.6.2). Since $\mathsf{consistent}(Q, L)$ is a separation logic proposition, these properties are stated using the separation logic connectives (and thus implicitly describe the threading and splitting of the heap typing Σ).

Lemma 3.7.2. *The buffer invariant is preserved by a sends, receives, and channel closure:*

- ⚙ *If* $L(p) = ![q]\{\ell : \tau_\ell. L_\ell\}_{\ell \in I}$, *then:*
  $(\emptyset \vDash v : \tau_\ell) * \mathsf{consistent}(Q, L) \twoheadrightarrow \mathsf{consistent}(\mathsf{push}(q, p, \langle \ell : \tau_\ell \rangle, Q), L[p := L_\ell])$.

- ⚙ *If* $L(q) = ?[p]\{\ell : \tau_\ell. L_\ell\}_{\ell \in I}$, *and* $\mathsf{pop}(q, p, \hat{Q}) = (\langle \ell : \tau \rangle, \hat{Q}')$, *then*
  $\mathsf{consistent}(Q, L) \twoheadrightarrow \ulcorner\ell \in I\urcorner * \mathsf{consistent}(Q', L[q := L_\ell]) * (\emptyset \vDash v : \tau_\ell)$.

- ⚙ *If* $L(p) = \mathsf{End}$, *then* $\mathsf{consistent}(Q, L) \twoheadrightarrow \mathsf{consistent}(Q \setminus \{p\}, L \setminus \{p\})$.

Lemma 3.7.3. ⚙ *If* $\mathsf{consistent}(\vec{L})$, *then* $\mathsf{Emp} \twoheadrightarrow \mathsf{consistent}(\mathsf{init}(\mathsf{length}(\vec{L})), \vec{L})$.

3.7.3 *The Configuration Invariant*

The invariant $\mathsf{wf}(\vec{e}, h)$ for configurations $(\vec{e}, h)$ ensures that every thread in $\vec{e}$ is well-typed, the contents of the buffers $h|_s$ for each session s in h are well-typed, the types of the channel literals match up with the types of the channels, and the communication topology is acyclic. To define this invariant, we instantiate the connectivity graph framework of Jacobs et al. (2022b) with the runtime typing judgment from Section 3.7.1 and the buffer invariant from Section 3.7.2.

The first ingredient of the connectivity framework is the data type $\mathsf{Cgraph(V, E)}$, which represents a directed graph with vertices ranging over the set V and edge labels ranging over the set E. This graph should be acyclic in an undirected sense (*i.e.,* the undirected erasure of the graph forms an undirected unrooted forest). We instantiate V and E in $\mathsf{Cgraph(V, E)}$ as follows:

$$\mathsf{V} ::= \mathsf{Thread}(i) \mid \mathsf{Session}(s) \qquad \mathsf{E} \triangleq \mathsf{Participant} \times \mathsf{LType} \qquad \clubsuit$$

The second ingredient of the connectivity graph framework is a generic invariant $\mathsf{wf(P)}$, which lifts a local invariant predicate $\mathsf{P}(v, \Delta) \in \mathsf{iProp}$ to whole runtime configurations. The local predicate P links the local configuration state of each vertex v (*i.e.,* the expression for a thread and the buffers for a session) to the multiset Δ of labels on the incoming edges of vertex v. Our instantiation $\mathsf{P}(v, \Delta) \triangleq \mathsf{wf}^{local}_{(\vec{e}, h)}(v, \Delta)$ is given in Figure 24. Intuitively, the local invariant for a thread (case $v = \mathsf{Thread}(i)$) says that the expression e_i of that thread is well-typed in the runtime type system with respect to the local types on the outgoing edges of the thread's vertex in the connectivity graph. The local invariant for a session (case $v = \mathsf{Session}(s)$) says that the buffers $h|_s$ of that session are well-typed with respect to the local types on the incoming edges of the session's vertex in the connectivity graph, where the endpoints stored in the buffers get their local types from the outgoing edges. The invariant for the whole configuration $\mathsf{wf}(\vec{e}, h)$ says that there exists an acyclic graph $G : \mathsf{Cgraph(V, E)}$ such that the local invariant predicate holds for all $v \in \mathsf{V}$.

3.7.4 *Initialization and Preservation of the Invariant*

The invariant holds for initial configurations and is preserved by the operational semantics:

Lemma 3.7.4. ♠ *If* $\emptyset \vdash e : \mathbf{1}$, *then* $\mathsf{wf}([e], \emptyset)$.

Lemma 3.7.5. ♠ *If* $(\vec{e}, h) \leadsto_{\mathrm{cfg}} (\vec{e}', h')$, *then* $\mathsf{wf}(\vec{e}, h)$ *implies* $\mathsf{wf}(\vec{e}', h')$.

The proof of the last lemma involves three aspects. First, because the configuration changes, we need to produce a connectivity graph for the new configuration as the connectivity graph is existentially quantified in the configuration invariant $\mathsf{wf}(\vec{e}, h)$. Second, we need to show that the new connectivity graph is acyclic in the appropriate sense. Third, we need to show that all the local invariant predicates $\mathsf{wf}^{local}_{(\vec{e}, h)}(v, \Delta)$ still hold. The interesting cases of this proof are the steps that involve the channel operations, for which the graph transformations are depicted in Figure 25.

Proving these graph transformations by picking a new graph by hand is cumbersome (especially in a mechanized proof). The connectivity graph framework therefore provides abstract separation logic lemmas to prove the transformations without having to mention the graph or having to deal with its acyclicity explicitly. We can re-use some of these abstract transformation rules, but for the n-ary **fork** we
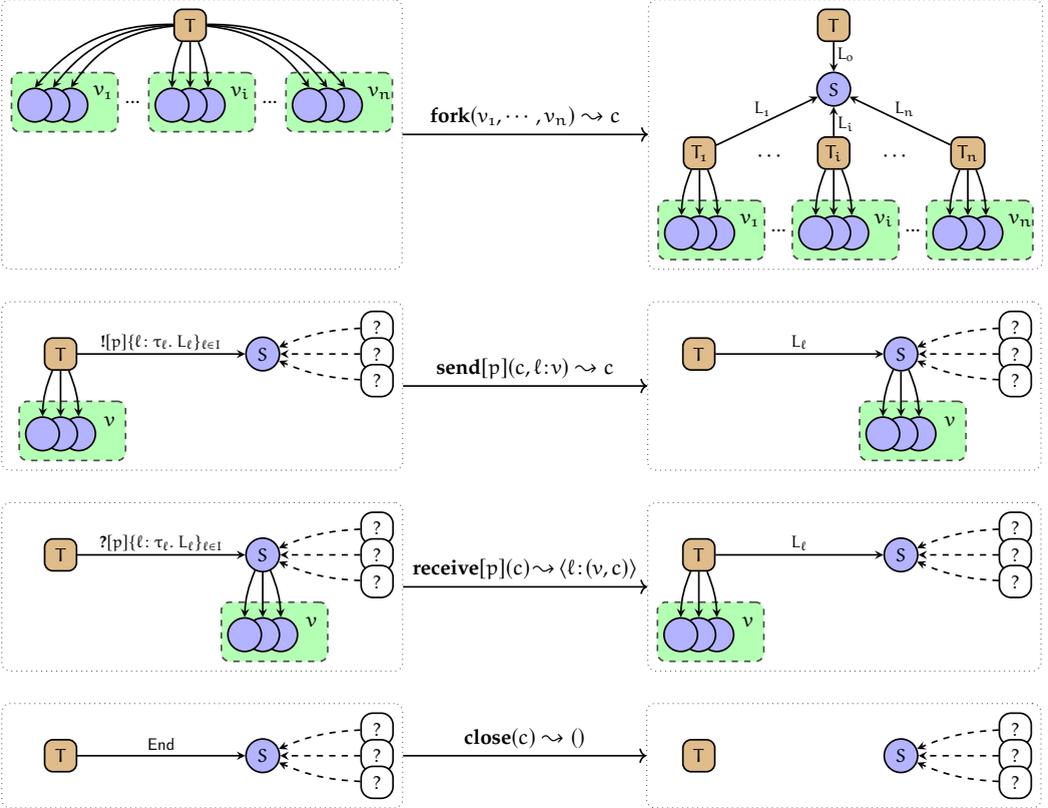
Figure 25: Graphical depiction of how multiparty interactions change the logical connectivity. Blue circles are multiparty sessions, brown squares are threads. A blue circle abstracts over the $n \times n$ communication paths among the n session participants, where each endpoint has buffers for incoming messages from every other endpoint. An edge from T to S indicates that thread T has an endpoint of session S. An edge from a session S_1 to a session S_2 indicates that an endpoint of S_2 is stored in one of the buffers of S_1. The figure provides a *local* viewpoint, only depicting the notions directly involved in an interaction and omitting other threads and sessions that are connected to the depicted ones as well. While the communication topology is cyclic within a multiparty session (where the global types rule out deadlock), it is acyclic *between* multiparty sessions, an invariant preserved by multiparty interactions. Acyclicity is crucial for deadlock and memory leak freedom.

need a new rule (which we state abstractly for arbitrary vertices V and edge labels E).

Lemma 3.7.6. ⚙ *Let $v_1, v_2 \in V$ and $w_1, \ldots, w_n \in V$. To prove* wf(P) *implies* wf(P'), *it suffices to prove, for all $\Delta \in$* Multiset E*:*

1. $P(v, \Delta) \twoheadrightarrow P'(v, \Delta)$ *for all $v \in V \setminus \{v_1, v_2, w_1, \ldots, w_n\}$*

2. $P(v, \Delta) \twoheadrightarrow \ulcorner \Delta = \emptyset \urcorner$ *for all $v \in \{v_2, w_1, \ldots, w_n\}$*

3. $P(v_1, \Delta) \twoheadrightarrow \exists l_0, \ldots, l_n. \ (\text{own}(v_2 \mapsto l_0) \twoheadrightarrow P'(v_1, \Delta)) *$
$$P'(v_2, \{l_0, \ldots, l_n\}) *$$
$$([*]i \in \{1..n\}. \ \text{own}(v_2 \mapsto l_i) \twoheadrightarrow P'(v_i, \emptyset))$$

3.7.5 *Proof of the Reachability Theorem*

We give an intuitive description of the proof of our main reachability theorem (Theorem 3.5.7 ⚙).

WAITING INDUCTION   At the top level of the proof, we apply the waiting induction principle of the connectivity graph library. Waiting induction relies on acyclicity of the graph and allows one to prove a predicate $P(v)$ for all vertices $v \in V$ of a graph $G : \text{Cgraph}(V, E)$, while assuming the "induction hypothesis" that $P(v')$ already holds for all vertices v' such that v is *waiting for* v', where *"waiting for"* is a binary relation chosen by us. The predicate $P(v)$ that we aim to prove for all vertices (*i.e.,* threads and sessions) is that v is reachable (see Theorem 3.5.7). The waiting relation we use is based on the $\text{blocked}_{(\vec{e}, h)}(v, v')$ relation, defined in Section 3.5. Waiting induction gives us the following induction hypotheses when proving that v is reachable:

FOR THREADS: If the thread is blocked on a session, we may assume that the session is reachable.

FOR SESSIONS: The owners of the session that are not blocked on this session are reachable.

REACHABILITY OF THREADS   To show that a thread is reachable, we must show that it can take a step, or that it is blocked on an endpoint of a session that is reachable. By typing, either the thread can take a pure step, or is a session operation where all subexpressions are values. A session operation can proceed if the structure of the heap is valid, which we can conclude from the configuration invariant. The only possibility for a blocked operation is if we are trying to receive and the buffer we are trying to receive from is currently empty. Here, the waiting induction hypothesis applies (because $\text{blocked}_{(\vec{e}, h)}$ holds), using which we can show that the session that we are blocked on is reachable. Then, by the definition of reachability, the thread is also reachable.

REACHABILITY OF SESSIONS   To show that a session s is reachable we must show that there exists a thread or session v that is (1) reachable, (2) holds an endpoint of s, and (3) is not blocked on s. We use the consistency of the buffers and local types of the session to show that there is an endpoint of s whose owner v is not blocked on this session (though v may be blocked on another session). This allows us to use the induction hypothesis to conclude that v is reachable (because $\text{blocked}_{(\vec{c},h)}(\mathsf{Session}(s), v)$ does *not* hold). Then, using the definition of reachability for sessions, we conclude that s is reachable.

MAIN RESULTS   Theorem 3.5.7 is obtained by combining the reasoning above with Theorem 3.7.4 and Theorem 3.7.5. Global progress (Theorem 3.5.1) follows as an easy corollary. For two directions of the equivalence of partial deadlock/leak freedom with full reachability (Theorem 3.5.6), we show that none of the objects in a deadlock/leak are reachable, and vice versa that the set of non-reachable threads and channels forms a deadlock/leak if this set is nonempty.

3.8 MECHANIZATION

All our results have been mechanized in Coq using Iris Proof Mode (Krebbers et al., 2017b, 2018) for the separation-logic part. The final results of our mechanization are Theorem 3.5.1, Theorem 3.5.6, and Theorem 3.5.7. We have also mechanized the translation from binary to multiparty in Figure 20 and have proved that it is type preserving. The mechanization is 10.4k lines of Coq code, consisting of 217 definitions, and 638 proved lemmas and theorems. Approximately half of the mechanization consists of results specific to our multiparty calculus, and the other half consists of the framework of Jacobs et al. (2022b), extended with support for n-ary graph transformations (Section 3.7.4).

PARTIAL DEADLOCK FREEDOM AND THE EMPTY TYPE   A surprising technical result of the mechanization is that while global progress remains true in the presence of n-ary sum types, we discovered that partial deadlock freedom is by default **false** for languages that allow $n = 0$. The reason is that a thread can throw away endpoints by pattern matching on the empty sum type. While this pattern match will never execute because the empty type can only be produced by a looping expression (thus guaranteeing global progress), a thread can still lose an endpoint during a substitution step *before* the empty pattern match happens. This can create a partial deadlock for the threads holding the other endpoints of the session, because they are now permanently blocked on a counterparty thread that has thrown away the endpoint. To fix this, we amend the typing judgment $\Gamma \vdash e : \tau$ to require the variable context Γ to be empty when pattern matching on an empty sum type. This formally ensures that the thread's expression keeps track of all endpoints and does not lose any. This does not limit the expressivity of empty types because one can obtain a

value of any type from an empty pattern match, including a function that can eat the remaining variables in the context.

3.9 RELATED WORK

To relate MPGV to the existing body of work it is helpful to consider two axes of categorization: *mechanization* and session type *philosophy*. The use of a proof assistant to mechanize correctness results has only been taken up recently by the session type community. Typeset pen-and-paper proofs and appeals to results in logic (*e.g.,* cut elimination) still constitute the status quo. We summarize mechanizations of session types below, but remark that only two works target mechanization of deadlock freedom up to date: Castro-Perez et al. (2021) for a single multiparty session and Jacobs et al. (2022b) for GV-style binary session types.

At first blush, session types can be distinguished into *binary* and *multiparty*. Whereas binary session types restrict the concurrent interaction to two participants, multiparty session types allow an arbitrary but statically determined number of participants ("roles"), by complementing the local perspective of a participant with a global type. A more foundational distinction, especially given the unifying nature of MPGV, is the underlying philosophy. Session types (Honda, 1993; Honda et al., 1998) have been conceived as a typing discipline for process calculi and as such preserve the fundamental characteristics of concurrent computation. Concurrent computation is inherently *non-deterministic* and may also give rise to *deadlocks*. For example, the below session-typed program from (Gay and Vasconcelos, 2010) (page 38) is well-typed but deadlocks:

$$\langle \text{let } c_1 = \text{request } a_1 \text{ in let } c_2 = \text{request } a_2 \text{ in let } (c_1, x) = \text{receive } c_1 \text{ in send } v \, c_2 \rangle \quad ||$$
$$\langle \text{let } d_1 = \text{accept } a_1 \text{ in let } d_2 = \text{accept } a_2 \text{ in let } (d_1, y) = \text{receive } d_2 \text{ in send } w \, d_1 \rangle$$

The program composes two threads (processes) in parallel, amounting to two binary interleaved sessions a_1 and a_2. Sessions are initiated by matching a request for a session (request a_1) with an accept for that session (accept a_1) creating two new endpoints per session (c_1 and c_2). The interleaving of the two sessions causes a deadlock: the receive on c_1 blocks the send on c_2, which is necessary for the former. The pairing of session requests with matching accepts is non-deterministic. For example, if we compose the two threads with a third thread that is also accepting a session a_1, then only one of the two accepting threads will be chosen.

This initialization pattern carries over to multiparty sessions (Honda et al., 2008, 2016). In the multiparty case a request is parameterized with the number of participants n and accepts with the role names ranging from 1 to $n-1$. Like binary session types, multiparty session types that assume this initialization pattern can deadlock. In particular, deadlocks can arise if a participant simultaneously engages in several sessions. A strategy adopted by some multiparty session type work (*e.g.,* Castro-Perez et al. (2021)) is to restrict a program to a single global multiparty session, precluding dynamic session spawning and first-class sessions. Alternatively,

advanced multiparty session-type systems (Coppo et al., 2013; Bettini et al., 2008; Coppo et al., 2016) employ extrinsic orders/priorities to rule out deadlocks among interleaved multiparty sessions, requiring order annotations in addition to global type declarations.

We refer to the line of session type work that adopts the initialization pattern shown above, which separates session creation from thread spawning, as *traditional* session types. We like to contrast this line of work with the one that adopts an initialization pattern based on cut, inspired by the Curry-Howard correspondence between linear logic and the session-typed π-calculus (Caires and Pfenning, 2010; Wadler, 2012; Lindley and Morris, 2015; Kokke et al., 2019), which we refer to as *logic-based* session types. Logic-based session types come with strong guarantees out of the box. These include, besides session fidelity, *deadlock freedom*. Given our focus on deadlock freedom, MPGV adopts the initialization pattern of logic-based session types and generalizes GV's fork construct (Wadler, 2012; Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021) for binary session types to the n-ary setting. The above program would thus not type check in MPGV.

A recent extension of GV by Fowler et al. (2021), Hypersequent GV (HGV), adopts hypersequents (Montesi and Peressotti, 2018; Kokke et al., 2019), inspired by Avron (1991), to facilitate a tighter correspondence to the session-typed π-calculus and simplify GV's meta theory by accounting for the forest topology of runtime structures. While this account is reminiscent of our notion of logical topology, HGV and our MPGV system are quite different. Firstly, HGV has binary session types, whereas our MPGV system supports multiparty session types. Secondly, HGV employs structural congruences for runtime typing, whereas our dynamics operates on a flat thread pool and heap (allowing an arbitrary thread to step without prior application of structural congruences) and our runtime typing relies on separation logic and connectivity graphs.

We next review the individual related work in more detail, referring to our categorization of traditional and logic-based session types as convenient. Given our focus on mechanization, we start with mechanized related work and then conclude with non-mechanized related work.

MECHANIZED   The only existing work on mechanizing deadlock freedom for multiparty session types is *Zooid*, a DSL by Castro-Perez et al. (2021) embedded in Coq. Although a traditional session type language in spirit, Zooid does neither support session spawning nor delegation, but restricts a program to a single global multiparty session. Zooid programs thus rule out deadlocks caused by multiparty session interleavings by construction. Thanks to a shallow embedding in Coq, Zooid programs can be extracted from Coq to OCaml via Coq's extraction mechanism. Send and receive operations are handled as monadic operations in which the endpoint is implicit (because there is a unique global session). A shallow embedding of binders works in this context because Zooid variables can only contain purely functional

data, which can be represented as Coq data. Our definition of (runtime) global types and projections is inspired by Zooid.

MPGV not only differs from Zooid in its support for multiple interacting sessions, first-class endpoints, dynamic spawning, and delegation, but also in statement and precision of the deadlock freedom property. Our mechanization guarantees global progress—including the stronger notions of partial deadlock/leak freedom. Zooid's main result, on the other hand, is phrased in terms of traces, asserting that for all traces produced by a well-typed process there *exists* a matching trace in the larger system. This result relies on properties of global types from the literature and also assumes the ability to choose a favorable schedule. Our mechanization in contrast states deadlock freedom for *all* executions/schedules and gives a closed end-to-end proof in Coq.

Jacobs et al. (2022b) contribute a mechanization of deadlock freedom for a variant of GV, and thus target *binary* session types. Like our mechanization, theirs accommodates dynamic session spawning and delegation, but restricted to the binary setting. Jacobs et al. (2022b) moreover contribute the notion of a *connectivity graph*, a parametric proof method for deadlock freedom, relying on acyclicity of the communication topology. We extend this proof method with n-ary operations and support of cyclic connectivity within a session governed by consistency. Our generalization to n-ary operations also enables our encoding of binary session types in MPGV (Section 3.4). Unlike Jacobs et al.'s variant of GV, our MPGV system moreover supports choice, and thus provides the first mechanization of deadlock and leak freedom for binary session types with choice.

Moreover, there exists work on mechanizing the metatheory of binary session types. Thiemann (2019) proves type safety of a linear λ-calculus with session types inspired by GV. The result does not include deadlock nor memory leak freedom. Hinrichsen et al. (2021) prove type safety for a comprehensive session-typed language with locks, subtyping and polymorphism using Iris in Coq. Their type system is affine, which means that deadlocks are considered safe. Tassarotti et al. (2017) prove termination preservation of a compiler for an affine session-typed language using Iris in Coq.

More distantly, there exist various mechanized results involving the π-calculus. Goto et al. (2016) prove type safety for a π-calculus with a polymorphic session type system in Coq. Their type system allows dropping channels, and hence does not enjoy deadlock nor memory leak freedom. Ciccone and Padovani (2020) mechanize dependent binary session types by embedding them into a π-calculus in Agda. They prove subject reduction (*i.e.,* preservation) and that typing is preserved by structural congruence, but not deadlock or memory leak freedom. Similarly, Zalakain and Dardha (2021) mechanize subject reduction of a resource-aware π-calculus, focusing on the handling of linear resources through leftover typing. Gay et al. (2020) study various notions of duality in Agda, and show that distribution laws for duality over the recursion operator are unsound. We adopted their approach of using coinductive types for mechanizing general recursive session types. Lastly, mechanizations

of choreographic languages (Montesi, 2021; Cruz-Filipe et al., 2021a,b) focus on determinism, confluence, and Turing completeness, with deadlock freedom holding by design.

NON-MECHANIZED   The work that is most closely related to ours in terms of underlying philosophy but non-mechanized is the work by Carbone et al. (2015, 2016, 2017) on coherence proofs. The authors introduce a proof theory grounded in classical linear logic via a Curry-Howard correspondence, illuminating the connection between binary and multiparty session types, in a π-calculus setting. The correspondence is due the novel notion of *coherence*, which generalizes duality known from binary session types to compatibility of local types with a global type of a multiparty session. Given a coherence derivation, an n-ary cut permits composing n participants concurrently, similar to our n-ary fork. Coherence also enables a semantic-preserving translation from multiparty sessions to corresponding binary sessions via an arbiter process (Carbone et al., 2016). Deadlock freedom follows from cut admissibility and cut elimination, giving a normalization argument. Such an argument is made possible by using cut reductions for the semantics and restricting to a non-Turing complete calculus without loops or recursion. In contrast, we provide a complete mechanization of deadlock freedom of an n-ary session-typed functional language, with recursive types, first-class endpoints, and a realistic asynchronous operational semantics based on an unstructured thread pool. Our encoding of binary sessions in MPGV moreover does not require an arbiter process.

Similarly, Caires and Pérez (2016) embed multiparty session types in a binary calculus by a translation from a global type to a *medium process*. Instead of communicating with each other, the participants communicate with the central medium process. This approach inherits deadlock freedom from the surrounding binary calculus, but requires central coordination and sequentializes the communication. Toninho and Yoshida (2019) show that the interconnection networks of classical linear logic (CLL) are strictly less expressive than those of a multiparty session calculus. *Partial multiparty compatibility* is used to define a new binary cut rule that can form circular interconnections but preserves the deadlock-freedom of CLL, albeit for a single multiparty session.

More distantly related are works that use Kobayashi-style type systems (Kobayashi, 1997, 2002a, 2006; Giachino et al., 2014; Kobayashi and Laneve, 2017) that enrich channel typing with usage information and partial orders to rule out cyclic dependencies among channel actions. In the traditional multiparty setting this is most notably the work by Coppo et al. (2013); Bettini et al. (2008); Coppo et al. (2016), which contributes an *interaction type system* that ensures deadlock freedom not only within but also between several multiparty sessions. This work not only differs from MPGV in that it requires ordering annotations in addition to global type declarations, but also in the statement of the global progress property. To account for processes that lack a communication partner, a possibility in the traditional setting, progress is stated relative to a catalyzing process that, if present, would allow the

closed program to step. MPGV sets itself additionally apart in its tight integration with a functional language.

Kobayashi-style systems have also been adopted for logic-based binary session types (Dardha and Gay, 2018; Kokke and Dardha, 2021c,a). The authors introduce a multicut, which allows for circular topologies within a session. To rule out deadlocks by type checking, session types must be annotated with *priorities*. Priority polymorphism has been used by Padovani (2014) to support cyclic interleavings of recursive processes. Partial order annotations, called *worlds*, are also used by Balzer et al. (2019), in a logic-based binary session type calculus that combines linear and shared (Balzer and Pfenning, 2017; Balzer et al., 2018) sessions. Shared session types introduce a controlled form of aliasing, an extension we would like to consider in future work.

A somewhat orthogonal approach to ensuring progress in the presence of dynamic thread allocation is to make global types more powerful. While traditional multiparty session types involve a fixed number of participants per session, Deniélou and Yoshida (2011); Demangeon and Honda (2012); Hu and Yoshida (2017) proposed extensions of single-session systems to make that number dynamic. This line of work does not support sessions as first-class data, and the expressivity is orthogonal to GV and MPGV. Hence, extending MPGV with a dynamic number of participants is an interesting extension for future work.

Chapter 4

A Self-Dual Distillation of Session Types

ABSTRACT   We introduce ⅄ ("lambda-barrier"), a minimal extension of linear λ-calculus with concurrent communication, which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed functional language also based on linear λ-calculus. Unlike GV, ⅄ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of session type duality. Instead, we use linear function function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.*, the function type constructor is dual to itself. Nevertheless, we can encode session types as ⅄ types, GV's channel operations as ⅄ terms, and show that this encoding is type-preserving. The linear type system of ⅄ ensures that all programs are deadlock-free and satisfy global progress, which we prove in Coq. Because of ⅄'s minimality, these proofs are simpler than mechanized proofs of deadlock freedom for GV.

4.1 INTRODUCTION

Session types (Honda et al., 1998; Honda, 1993) are types for communication channels, that can be used to verify that programs follow the communication protocol specified by a channel's session type. Gay and Vasconcelos (Gay and Vasconcelos, 2010) embed session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus (Gay and Vasconcelos, 2010) did not yet ensure deadlock freedom, Wadler's subsequent GV (Wadler, 2012) and its derivatives (Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021) guarantee that all well-typed programs are deadlock free.

In order to add session types to linear λ-calculus, one adds (linear) session type formers for typing channel protocols and their corresponding operations: $!\tau.s$ (send a message of type τ, continue with protocol s), $?\tau.s$ (receive a message of type τ, continue with protocol s), $s_1 \oplus s_2$ (send choice between protocols s_1 and s_2), $s_1 \mathbin{\&} s_2$ (receive choice between protocols s_1 and s_2), and End (close channel). One also adds a **fork** operation for creating a thread and a pair of dual channels. For this, we need a definition of duality, with ! dual to ?, $\oplus$ dual to &, and End dual to itself.

There have been efforts for simpler systems, such as an encoding of session types into ordinary π-calculus types (Kobayashi, 2002b; Dardha et al., 2012, 2017), and *minimal session types* (Arslanagic et al., 2019), which decompose multi-step session types into single-step session types in a π-calculus. Single-shot synchronization

primitives have also been used in the implementation of a session-typed channel libraries (Scalas and Yoshida, 2016a; Padovani, 2017; Kokke and Dardha, 2021a).

We show that linear λ-calculus is also an excellent substrate on which to build a minimal concurrent calculus with communication, and introduce ⅄ ("lambda-barrier"), which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed functional language that is also based on linear λ-calculus. Unlike GV, ⅄ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of duality. Instead, we use the linear function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.,* the function type constructor is dual to itself. Nevertheless, we can encode session types as ⅄ types, GV's channel operations as ⅄ terms, and show that this encoding is type-preserving. A key difference with CPS encodings of GV (Lindley and Morris, 2016b,c), which are whole-program, is that our encoding is local, and uses ⅄'s built-in concurrency.

Like GV, all well-typed ⅄ programs are automatically *deadlock free*, and therefore satisfy *global progress*. We prove this property in Coq. Because of ⅄'s minimality, these proofs are simpler and shorter than mechanized proofs of deadlock freedom for GV.

The rest of this chapter is structured as follows:

- An introduction to ⅄ by example (Section 4.2).

- The ⅄ type system and operational semantics (Section 4.3).

- Encoding session types in ⅄ (Section 4.4).

- How to prove global progress and deadlock freedom for ⅄ (Section 4.5).

- Extending ⅄ with unrestricted and recursive types (Section 4.6).

- Mechanizing the meta-theory of ⅄ in Coq (Section 4.7).

- Related work (Section 4.8).

- Concluding remarks (Section 4.9).

4.2 THE ⅄ LANGUAGE BY EXAMPLE

The ⅄ language consists of linear λ-calculus with a single extension: **fork**.[1] Let us look at an example:

$$\textbf{let } x = \textbf{fork}(\lambda x'. \textbf{ print}(x' \; 1)) \textbf{ in print}(1 + x \; 0)$$

This program forks off a new thread, which also creates *communication barriers* x and x' to communicate between the threads. The barrier x gets returned to the main

---

1 For the examples we also use **print**, to be able to talk about the operational behavior of programs.

thread, and x′ gets passed to the child thread. These barriers are functions, and a call to a barrier will block until the other side is also trying to synchronize, and will then atomically exchange the values passed as an argument. The example runs as follows:

- When x′ 1 is called, it will block until x 0 is also called, and vice versa.

- The call x′ 1 will then return 0, and the call x 0 will return 1.

Thus, the program will print 0 2 or 2 0, depending on which thread prints first. In ⅄, these barriers are *linear*, so they must be used exactly once:

$$\textbf{fork}(\lambda x.\ \textbf{print}(1)) \quad \text{Error! Must use x.}$$
$$\textbf{fork}(\lambda x.\ \textbf{print}(x\ 0 + x\ 1)) \quad \text{Error! Can't use x twice.}$$

The type of **fork** is:

$$\textbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \textbf{1}) \to (\tau_2 \multimap \tau_1)$$

where $\multimap$ is the type of linear functions. Linearity allows us to encode *session types* in ⅄ (Section 4.4), and ensures that all well-typed ⅄ programs are *deadlock-free* (Section 4.5), which would not be the case without linearity. Nevertheless, linearity may seem like a critical limitation: can a child thread communicate with its parent thread only *once*?! Luckily, two features of λ-calculus, namely the ability for closures to capture values from their lexical environment, and the ability to pass functions as arguments to other functions, means that the restriction is not as severe as it may seem. Let's look at an example that uses those two features:[2]

$$\textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1))$$
$$\textbf{let } y = \textbf{fork}(\lambda y'.\ y'\ x)$$
$$\textbf{print}(1 + y\ ()\ 0)$$

We fork off a new thread (line 1) and store its barrier in x. We then fork off another thread (line 2), and pass the barrier x into the λ-expression of the new thread. We call y (), which returns x, because the thread of y calls y′ x. Finally, we pass 0 into the returned x, so this example behaves the same as the first example: it prints 0 2 or 2 0.

---

2 We omit the **in** keyword if a newline follows **let**, like some functional languages (*e.g.,* F#).

We can use the ability to capture barriers in the λ-expression of a fork, and the ability to send barriers over barriers, to set up long-running communication between two threads:

$$\textbf{let } x_1 = \textbf{fork}(\lambda x_1'.\ \textbf{let } (x_2', n_1) = x_1'\ ()$$
$$\textbf{let } (x_3', n_2) = x_2'\ ()\ \textbf{in } x_3'\ (n_1 + n_2))$$
$$\textbf{let } x_2 = \textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1))$$
$$\textbf{let } x_3 = \textbf{fork}(\lambda x_3'.\ x_2\ (x_3', 2))$$
$$\textbf{print}(x_3\ ())$$

Let us focus on the body of the first fork. The forked thread firstly synchronizes with its barrier, via $x_1'\ ()$. This call will return a pair (x_2', n_1) of a *new* barrier x_2', and a number n_1. It then synchronizes with the new barrier, via $x_2'\ ()$, which returns *another* pair (x_3', n_2), giving yet another barrier x_3' and another number n_2. In the last step, it sends the number $n_1 + n_2$ back to the main thread, via $x_3'\ (n_1 + n_2)$.

Let us now focus on how the main thread arranges this sequence of communications. The main thread first forks off a *messenger thread* $\textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1))$. The purpose of this thread is to send the message $(x_2', 1)$ over x_1, where x_2' is the barrier associated with the messenger thread. The other side of that barrier, x_2, is given to the main thread. The main thread now forks off another messenger thread, this time using that new barrier, x_2. This gives the main thread yet another barrier, x_3, from which it receives the final answer, $1 + 2$, via $x_3\ ()$.

Note that, like in the asynchronous π-calculus, sending a message involves forking off a tiny thread. Thus, like the asynchronous π-calculus, λ should be viewed as a theoretical core calculus, and not as a practical way to implement message passing.[3] We can encapsulate this messenger thread pattern in a small library of channel operations:

$$\textbf{send}(c, x) \triangleq \textbf{fork}(\lambda c'.\ c\ (c', x))$$
$$\textbf{receive}(c) \triangleq c\ ()$$
$$\textbf{close}(c) \triangleq c\ ()$$

---

3 Because the messenger threads are always of a specific form, it might be possible to implement a compiler that recognizes such patterns and implements them more efficiently. After all, the messenger threads do nothing but immediately synchronize with another barrier.

Using this channel library, we can implement the preceding example in the following way:

$$
\begin{aligned}
\textbf{let } x_1 = \textbf{fork}(\lambda x_1'. \ &\textbf{let } (x_2', n_1) = \textbf{receive}(x_1') \\
&\textbf{let } (x_3', n_2) = \textbf{receive}(x_2') \\
&\textbf{let } x_4' = \textbf{send}(x_3', n_1 + n_2) \\
&\textbf{close}(x_4')) \\
\textbf{let } x_2 = \ &\textbf{send}(x_1, 1) \\
\textbf{let } x_3 = \ &\textbf{send}(x_2, 2) \\
\textbf{let } (x_4, n) = \ &\textbf{receive}(x_3) \\
\textbf{print}&(n) \\
\textbf{close}&(x_4)
\end{aligned}
$$

Session-typed channels usually also have *choice*, which allows choosing between two continuation protocols. This can be encoded in λ using sums $\textbf{in}_L(x)$ and $\textbf{in}_R(x)$:

$$
\begin{aligned}
\textbf{tell}_L(c) &\triangleq \textbf{fork}(\lambda c'. \ c \ \textbf{in}_L(c')) \\
\textbf{tell}_R(c) &\triangleq \textbf{fork}(\lambda c'. \ c \ \textbf{in}_R(c')) \\
\textbf{ask}(c) &\triangleq c \ ()
\end{aligned}
$$

With these operations, we can implement the calculator example of Lindley and Morris (Lindley and Morris, 2017). This example allows the client to choose whether they want to add two numbers or negate a number. If the client chooses to add two numbers, they then send two numbers as separate messages, and retrieve the answer using receive. If the client chooses to negate a number, they then send only a single number, and retrieve the answer using receive. This example illustrates the choice between two different protocols for the remaining interaction:

let calc c =

    match ask(c) **with**

   | $\textbf{in}_L(c) \Rightarrow$ **let** $(c, n) = \textbf{receive}(c)$

            let $(c, m) = \textbf{receive}(c)$

            close(**send**(c, n + m))

   | $\textbf{in}_R(c) \Rightarrow$ **let** $(c, n) = \textbf{receive}(c)$

            close(**send**(c, −n))

  end

Extending ⅄ with recursion (Section 4.6) allows us to implement unbounded protocols, as illustrated by the following example:

let rec countdown c =

    match ask(c) **with**

    | **in$_L$**(c) $\Rightarrow$ **close**(c)

    | **in$_R$**(c) $\Rightarrow$ **let** (c, n) = **receive**(c)

               print(n)

               if n = 0

                then close(**tell$_L$**(c))

                else countdown (**send**(**tell$_R$**(c), n − 1))

    end

Given a channel c, the countdown c program first uses **ask**(c) to ask c if it wants to terminate, and closes the channel if so. Otherwise it receives a number n from c and prints it. Then it checks if the number n = 0, and if so tells the other side to close (using **tell$_L$**), and then closes our side. Otherwise it tells the other side that it wants to continue (using **tell$_R$**) and sends n − 1 to the other side. We can therefore let countdown interact with a copy of itself, provided we start off one of the copies with an initial message:

countdown **send**(**tell$_R$**(**fork**(countdown)), 10)

This program will print the numbers 10 9 8 $\cdots$ 1 0, in that order. The odd numbers are printed by the main thread, and the even numbers are printed by the child thread.

As we shall see later, this is all type-safe. If we had not started off one of the countdowns with an initial message, and had instead simply done countdown **fork**(countdown), then we would have had a static type error. The reason is that the first action of countdown c is to perform **ask**(c), which requires c to be of type () $\multimap$ τ, where τ is the type of the message sent by the child thread. The type of fork allows τ_1 $\multimap$ τ_2 to interact with τ_2 $\multimap$ τ_1. Therefore, τ would have to be () in order to make () $\multimap$ τ interact with itself, but the message type τ is a sum type in this case, and thus not ().

This way, the ⅄ type system ensures that channel protocols are correctly followed, even though the ⅄ type system has no session types and no notion of duality and instead simply uses function types τ_1 $\multimap$ τ_2 for barriers. We do not need an explicit notion of duality because the function type constructor is *self-dual*, in the sense that if x : τ_1 $\multimap$ τ_2 is a barrier, then the dual barrier with which x will synchronize has type x' : τ_2 $\multimap$ τ_1. We will see more about encoding session types in ⅄ in Section 4.4.

$$\frac{\cdot}{x{:}\tau \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 : \tau_3}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{in_L}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{in_R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma_1 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau' \qquad \Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau'}{\Gamma_1, \Gamma_2 \vdash \mathbf{match}\ e\ \mathbf{with\ in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} : \tau'}$$

Figure 26: Linear λ-calculus with sums and products (rules for **0** and **1** omitted).

4.3 THE ⋏ TYPE SYSTEM AND OPERATIONAL SEMANTICS

Like GV (Wadler, 2012) and its derivatives (Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021), the basis of ⋏ is a linear simply typed λ-calculus. We have sums, products, and the linear function type $\tau_1 \multimap \tau_2$. Variables of linear type must be used exactly once: they cannot be duplicated (contracted) or discarded (weakened), so that one must use one of the elimination rules of the type.

$$\tau \in \mathsf{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau \mid \tau \to \tau$$

Our basis linear λ-calculus has the following grammar of expressions, which consists of introduction and elimination forms for each type:

$$e \in \mathsf{Expr} ::= x \mid () \mid (e, e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\, e \mid e\, e \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid$$
$$\mathbf{match}\ e\ \mathbf{with\ end} \mid \mathbf{match}\ e\ \mathbf{with\ in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end}$$

The typing rules are standard and can be found in Figure 26.

We now have a substrate to which we will add concurrency constructs. GV (Wadler, 2012; Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021) introduces concurrency by means of a construct to spawn a new thread, with which we can communicate using a channel. Communication is governed by session types, such that the two endpoints of a channel are typed with dual session types. Instead, ⋏ extends the substrate with concurrency in a minimal way, adding one new construct to create new threads:

$$e \in \mathsf{Expr} ::= \cdots \mid \mathbf{fork}(e)$$

This is the typing rule for **fork**:

$$\frac{\Gamma \vdash e : (\tau_2 \multimap \tau_1) \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}(e) : \tau_1 \multimap \tau_2}$$

Or, as a type signature:

$$\mathbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \mathbf{1}) \rightarrow (\tau_2 \multimap \tau_1)$$

The type of **fork** uses the linear function type. We do not need an explicit notion of duality, like session types do, because the function type constructor is *self-dual*, in the sense that if $x : \tau_1 \multimap \tau_2$ is a barrier, then the dual barrier x' with which x will synchronize has type $x' : \tau_2 \multimap \tau_1$.

4.3.1 *Operational semantics*

We use a small-step operational semantics with evaluation contexts. In order to represent barriers, we add barrier literals $\langle k \rangle$, $k \in \mathbb{N}$ to the expressions. A barrier literal cannot appear in the source program, as the static type system has no typing rule for it. Barrier literals only appear in expressions at runtime when the operational semantics executes a **fork**-step. This gives us the following set of values for the language:

$$v \in \mathsf{Val} ::= () \mid (v, v) \mid \mathbf{in_L}(v) \mid \mathbf{in_R}(v) \mid \lambda x.\ e \mid \langle k \rangle$$

We have four pure head-reduction rules $e \leadsto_{\mathrm{pure}} e'$, one for λ, one for pairs, and two for sums, as stated in Figure 27. We use evaluation contexts to avoid introducing many congruence rules:[4]

$$K ::= \square \mid (K, e) \mid (v, K) \mid \mathbf{in_L}(K) \mid \mathbf{in_R}(K) \mid K\ e \mid v\ K \mid \mathbf{fork}(K) \mid \mathbf{let}\ (x_1, x_2) = K\ \mathbf{in}\ e$$
$$\mid \mathbf{match}\ K\ \mathbf{with}\ \ \mathbf{end} \mid \mathbf{match}\ K\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end}$$

We represent a configuration with multiple threads and barriers as a finite map:

$$\rho \in \mathsf{Cfg} \triangleq \mathbb{N} \xrightarrow{\mathrm{fin}} \mathsf{Thread(Expr)} + \mathsf{Barrier}$$

We define a **configuration step relation** $\rho \overset{i}{\leadsto} \rho'$. The label $i \in \mathbb{N}$ is used to keep track of which thread or barrier in the configuration takes the step. This has no effect on the operational semantics, but we will later use it to formulate deadlock freedom. The rules for the configuration step relation are given in Figure 27. We have the following five rules, in the order of the figure:

---

4 This set of evaluation contexts results in left-to-right evaluation order, but the mechanization has been set up so that the proof scripts work for right-to-left and nondeterministic order as well.

$$(\lambda x.\ e)\ v \leadsto_{\text{pure}} e[v/x]$$

$$\textbf{let }(x_1, x_2) = (v_1, v_2) \textbf{ in } e \leadsto_{\text{pure}} e[v_1/x_1][v_2/x_2]$$

$$\textbf{match in}_L(v) \textbf{ with in}_L(x_1) \Rightarrow e_1 \mid \textbf{in}_R(x_2) \Rightarrow e_2 \textbf{ end} \leadsto_{\text{pure}} e_1[v/x_1]$$

$$\textbf{match in}_R(v) \textbf{ with in}_L(x_1) \Rightarrow e_1 \mid \textbf{in}_R(x_2) \Rightarrow e_2 \textbf{ end} \leadsto_{\text{pure}} e_2[v/x_2]$$

$$\left\{ n \mapsto \text{Thread}(K[\ e_1\ ]) \right\} \stackrel{n}{\leadsto} \left\{ n \mapsto \text{Thread}(K[\ e_2\ ]) \right\} \quad \text{if } e_1 \leadsto_{\text{pure}} e_2 \qquad \text{(pure)}$$

$$\left\{ n \mapsto \text{Thread}(K[\ \textbf{fork}(v)\ ]) \right\} \stackrel{n}{\leadsto} \begin{cases} n \mapsto \text{Thread}(K[\ \langle k\rangle\ ]) \\ k \mapsto \text{Barrier} \\ m \mapsto \text{Thread}(v\ \langle k\rangle) \end{cases} \qquad \text{(fork)}$$

$$\begin{cases} n \mapsto \text{Thread}(K_1[\ \langle k\rangle\ v_1\ ]) \\ k \mapsto \text{Barrier} \\ m \mapsto \text{Thread}(K_2[\ \langle k\rangle\ v_2\ ]) \end{cases} \stackrel{k}{\leadsto} \begin{cases} n \mapsto \text{Thread}(K_1[\ v_2\ ]) \\ m \mapsto \text{Thread}(K_2[\ v_1\ ]) \end{cases} \qquad \text{(sync)}$$

$$\left\{ n \mapsto \text{Thread}(()) \right\} \stackrel{n}{\leadsto} \left\{ \right\} \qquad \text{(exit)}$$

$$\rho_1 \uplus \rho' \stackrel{i}{\leadsto} \rho_2 \uplus \rho' \quad \text{if } \rho_1 \stackrel{i}{\leadsto} \rho_2 \quad (\uplus \text{ is disjoint union)} \quad \text{(frame)}$$

Figure 27: The operational semantics of λ.

(PURE) A rule for pure reductions for a single thread.

(FORK) A rule for forking a new thread, which adds the new thread and a barrier k to the configuration. The two threads get access to the barrier via the barrier literal $\langle k\rangle$.

(SYNC) A rule to synchronize on a barrier. The two threads that are synchronizing exchange the values v_1 and v_2. This step removes the barrier.

(EXIT) A rule for removing finished threads from the configuration.

(FRAME) A rule for extending the preceding rules to larger configurations, by allowing the rest of the configuration to pass through unchanged.

This is a possible execution of the second example from Section 4.2:

$$\left\{ 0 \mapsto \text{Thread} \begin{pmatrix} \textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1)) \\ \textbf{let } y = \textbf{fork}(\lambda y'.\ y'\ x) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \right\} \overset{0}{\leadsto}$$

$$\left\{ \begin{array}{l} 0 \mapsto \text{Thread} \begin{pmatrix} \textbf{let } y = \textbf{fork}(\lambda y'.\ y'\ \langle 1 \rangle) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \\ 1 \mapsto \text{Barrier} \\ 2 \mapsto \text{Thread} (\textbf{print}(\langle 1 \rangle\ 1)) \end{array} \right\} \overset{0}{\leadsto} \left\{ \begin{array}{l} 0 \mapsto \text{Thread} \left(\textbf{print}(1 + \langle 3 \rangle\ ()\ 0) \right) \\ 1 \mapsto \text{Barrier} \\ 2 \mapsto \text{Thread} (\textbf{print}(\langle 1 \rangle\ 1)) \\ 3 \mapsto \text{Barrier} \\ 4 \mapsto \text{Thread} (\langle 3 \rangle\ \langle 1 \rangle) \end{array} \right\} \overset{3}{\leadsto}$$

$$\left\{ \begin{array}{l} 0 \mapsto \text{Thread} \left(\textbf{print}(1 + \langle 1 \rangle\ 0) \right) \\ 1 \mapsto \text{Barrier} \\ 2 \mapsto \text{Thread} (\textbf{print}(\langle 1 \rangle\ 1)) \\ 4 \mapsto \text{Thread} (()) \end{array} \right\} \overset{4}{\leadsto} \left\{ \begin{array}{l} 0 \mapsto \text{Thread} \left(\textbf{print}(1 + \langle 1 \rangle\ 0) \right) \\ 1 \mapsto \text{Barrier} \\ 2 \mapsto \text{Thread} (\textbf{print}(\langle 1 \rangle\ 1)) \end{array} \right\} \overset{1}{\leadsto}$$

$$\left\{ \begin{array}{l} 0 \mapsto \text{Thread} \left(\textbf{print}(1 + 1) \right) \\ 2 \mapsto \text{Thread} (\textbf{print}(0)) \end{array} \right\} \overset{2*}{\leadsto} \left\{ 0 \mapsto \text{Thread} \left(\textbf{print}(1 + 1) \right) \right\} \overset{1*}{\leadsto} \{\}$$

For simplicity, we treat **print** on natural numbers as a no-op that returns (), instead of adding an output log to the semantics, because whether or not **print** logs its output somewhere does not affect the further execution of the program.

While untyped ƛ programs can easily get stuck, for instance if one side throws away its barrier, or sets up cyclic waiting dependencies, well-typed ƛ programs never get stuck. More formally, we prove *global progress* (in Section 4.5), which means that if we start with an initial program $e : \mathbf{1}$, then any non-empty configuration we can reach from e can step further. But first, we will encode session types in ƛ.

4.4 ENCODING SESSION TYPES IN ƛ

Despite being very simple, ƛ's type system can encode session types. There are five basic session type constructors:

$$s \in \text{Session} \triangleq \text{End} \mid \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \oplus s \mid s \mathbin{\&} s$$

The type $\mathbf{!}\tau.s$ means to send a value of type τ and then continue with s. Dually, $\mathbf{?}\tau.s$ means to receive a value of type τ and then continue with s. The type $s_1 \oplus s_2$ indicates that we have a choice of continuing either with protocol s_1 or with s_2. Dually, the type $s_1 \mathbin{\&} s_2$ means that we receive a choice from the other side: either we have to continue with protocol s_1 or with protocol s_2, depending on what the other side chose. Lastly, the protocol End means that we are done with the channel

and we must deallocate it. Session types make the notion of duality explicit using the function dual : Session → Session:

$$dual(End) \triangleq End$$
$$dual(!\tau.s) \triangleq ?\tau.dual(s)$$
$$dual(?\tau.s) \triangleq !\tau.dual(s)$$
$$dual(s_1 \oplus s_2) \triangleq dual(s_1) \& dual(s_2)$$
$$dual(s_1 \& s_2) \triangleq dual(s_1) \oplus dual(s_2)$$

The idea is that if our channel has type s, then the channel of the party we are communicating with has type $dual(s)$. This is the list of channel operations and their types:

fork$_{GV}$: $(s \multimap \mathbf{1}) \rightarrow dual(s)$

Fork off a new thread running the closure. Passes a channel of type s to the child thread, and returns a channel of type $dual(s)$ to the main thread.

close : $End \rightarrow \mathbf{1}$

Close and deallocate the channel. Returns unit ().

send : $!\tau.s \times \tau \rightarrow s$

Send a message of type τ to the channel. Returns a new channel of type s for performing the rest of the protocol.

receive : $?\tau.s \rightarrow s \times \tau$

Receive a message from the channel. Returns a pair $s \times \tau$ of the channel for performing the rest of the protocol (type s) and the message received (type τ).

tell$_L$: $s_1 \oplus s_2 \rightarrow s_1$

In a branching protocol, choose the left branch. Returns a channel of the chosen type.

tell$_R$: $s_1 \oplus s_2 \rightarrow s_2$

In a branching protocol, choose the right branch. Returns a channel of the chosen type.

ask : $s_1 \& s_2 \rightarrow s_1 + s_2$

Receives the choice made by the other side. Returns a sum type, which is **in$_L$**(c) with c : s_1 if the left branch was chosen by the other side, and **in$_R$**(c) with c : s_2 if the right branch was chosen.[5]

We will encode channels as λ̃'s barriers, and we therefore encode a session type s as a linear function type $\tau_1 \multimap \tau_2$ where τ_1, τ_2 are determined from s. Intuitively, the sending side not only transfers the values specified by the protocol, but also a

---

5 In the original GV, branching was combined with receiving a choice. We decouple them, and let receiving a choice return a sum type, which can subsequently be pattern matched on using **match**.

continuation channel for the remainder of the protocol. The continuation channel is connected to a tiny messenger thread, which is responsible for synchronizing with the old barrier, as we did in Section 4.2. We define an encoding function $[\![\cdot]\!]$: Session → Type that converts a session type to a λ type. The encoding of session types into λ types is as follows:

$$[\![End]\!] \triangleq \mathbf{1} \multimap \mathbf{1}$$
$$[\![!\tau.s]\!] \triangleq [\![dual(s)]\!] \times \tau \multimap \mathbf{1}$$
$$[\![?\tau.s]\!] \triangleq \mathbf{1} \multimap [\![s]\!] \times \tau$$
$$[\![s_1 \oplus s_2]\!] \triangleq [\![dual(s_1)]\!] + [\![dual(s_2)]\!] \multimap \mathbf{1}$$
$$[\![s_1 \,\&\, s_2]\!] \triangleq \mathbf{1} \multimap [\![s_1]\!] + [\![s_2]\!]$$

Using this encoding, we can implement channel operations with type signatures matching their native session-typed version, provided we use the encoding:

$$
\begin{aligned}
\mathbf{fork}_{GV} &: ([\![s]\!] \multimap \mathbf{1}) \to [\![dual(s)]\!] & \mathbf{fork}_{GV}(x) &\triangleq \mathbf{fork}(x) \\
\mathbf{close} &: [\![End]\!] \to \mathbf{1} & \mathbf{close}(c) &\triangleq c\,() \\
\mathbf{send} &: [\![!\tau.s]\!] \times \tau \to [\![s]\!] & \mathbf{send}(c,x) &\triangleq \mathbf{fork}(\lambda c'.\, c\,(c',x)) \\
\mathbf{receive} &: [\![?\tau.s]\!] \to [\![s]\!] \times \tau & \mathbf{receive}(c) &\triangleq c\,() \\
\mathbf{tell}_L &: [\![s_1 \oplus s_2]\!] \to [\![s_1]\!] & \mathbf{tell}_L(c) &\triangleq \mathbf{fork}(\lambda c'.\, c\,\mathbf{in}_L(c')) \\
\mathbf{tell}_R &: [\![s_1 \oplus s_2]\!] \to [\![s_2]\!] & \mathbf{tell}_R(c) &\triangleq \mathbf{fork}(\lambda c'.\, c\,\mathbf{in}_R(c')) \\
\mathbf{ask} &: [\![s_1 \,\&\, s_2]\!] \to [\![s_1]\!] + [\![s_2]\!] & \mathbf{ask}(c) &\triangleq c\,()
\end{aligned}
$$

The fork operation for channels simply delegates to the fork operation of λ, because a channel is represented as a barrier.

FORMAL STATEMENT OF WELL-TYPEDNESS OF THE ENCODINGS   You may note that while there is an encoding function $[\![\cdot]\!]$ of session types into λ types, there is no explicit encoding function of GV terms to λ terms. This is intentional: because the translation is *local*, the definitions above can be viewed as *syntactic abbreviations* or *macros*. For instance, we can define the abbreviation

$$\mathbf{tell}_L \triangleq \lambda c.\, \mathbf{fork}(\lambda c'.\, c\,\mathbf{in}_L(c'))$$

of the $\mathbf{tell}_L$ channel operation as a closed syntactic λ term. We can then *prove* that for all session types s_1 and s_2, the typing judgement

$$\emptyset \vdash \mathbf{tell}_L : [\![s_1 \oplus s_2]\!] \to [\![s_1]\!]$$

for the **tell$_L$** term given above is derivable from λ's typing rules. The most interesting case is **fork**; in order to prove

$$\emptyset \vdash \textbf{fork} : ([\![s]\!] \multimap \textbf{1}) \rightarrow [\![\text{dual}(s)]\!]$$

for all session types s, we rely on the following lemma about dual and the encoding $[\![\cdot]\!]$:

$$([\![s]\!] = \tau_1 \multimap \tau_2) \iff ([\![\text{dual}(s)]\!] = \tau_2 \multimap \tau_1)$$

That is, if the session types are dual in the session types sense, then their encodings are dual in the λ sense.

The advantage of this approach is its simplicity and that we can freely intermix channels with direct usage of barriers in the same program. However, for our simulation result (Section 4.4.1), we do need an explicit definition of GV syntax and a translation of GV terms to λ terms.

A NOTE ON THE MECHANIZATION AND n-ARY CHOICE   We can combine n-ary choice with sending/receiving a message in a single communication step using n-ary sum types:

$$\textbf{sendchoice}_i : !\{i : \tau_i.s_i\}_{i \in I} \times \tau_i \rightarrow s_i$$
$$\textbf{receivechoice} : ?\{i : \tau_i.s_i\}_{i \in I} \rightarrow \Sigma_{i \in I} \, s_i \times \tau_i$$

This can also be encoded in λ, and is what is provided by the mechanization (4.7):

$$\textbf{sendchoice}_i(c, x) \triangleq \textbf{fork}(\lambda c'. \, c \, \textbf{in}_i(c', x))$$
$$\textbf{receivechoice}(c) \triangleq c \, ()$$

ENCODING λ IN GV   We can also do the encoding the other way around, and implement λ's fork in terms of GV's channel constructs:

fork$_\lambda$(f) $\triangleq$

  let $c_1 = \textbf{fork}_{GV}(\lambda c_1'. \, f \, (\lambda v'. \, \textbf{let} \, c_2' = \textbf{send}(c_1', v')$

                                  let $(c_3', v) = \textbf{receive}(c_2')$

                                    close$(c_3'); \, v))$

  $\lambda v. \, \textbf{let} \, (c_2, v') = \textbf{receive}(c_1)$

      let $c_3 = \textbf{send}(c_2, v)$

      close$(c_3); \, v'$

Given how short the encodings of GV's channel operations in λ are, it is perhaps surprising that the other way around requires comparatively more code.

4.4.1 *Simulation of GV's semantics with ⅄'s semantics*

To show that the encoding makes sense, we prove that we can simulate an asynchronous version of GV using ⅄'s semantics. We use an asynchronous semantics ($\leadsto_{GV}$) for GV, so the GV configuration contains buffers. For details about the GV semantics used in the proof, we refer the reader to the mechanization (Section 4.7). The key idea behind the simulation is that each message in a buffer on the GV side corresponds to a messenger thread on the ⅄ side. Whenever a message is put in a buffer on the GV side, a messenger thread is created on the ⅄ side, and the messenger thread will be waiting to synchronize with a barrier. Whenever a message is received from a channel's buffer on the GV side, the receiver and the messenger thread execute their sync operation on the ⅄ side, which sends the message to the receiver and allows the messenger thread to terminate.

Formally, we start with an encoding ⟦e⟧ that translates GV terms to the corresponding ⅄ terms by replacing all occurrences of GV channel operations with their ⅄ definitions given above. We then extend this translation to configurations ⟦ρ⟧, The translation on configurations replaces each buffer in the GV heap with a sequence of ⅄ messenger threads, with one messenger thread per message in the buffer. With these notions at hand, we can show that the ⅄ encodings simulate the GV semantics.

Lemma 4.4.1 (Simulation). *If $\rho \leadsto_{GV} \rho'$ then ⟦ρ⟧ $\leadsto^*$ ⟦ρ'⟧*

This lemma has been mechanized in Coq (Section 4.7). We need the transitive closure ($\leadsto^*$) on the ⅄ side, because a single step in the GV semantics can correspond to multiple steps in the ⅄ semantics, since the ⅄ semantics does extra administrative β-reductions. For instance, when the GV program does a **send**(c, v) operation, it places the message v in the buffer in one step. The translated ⅄ program on the other hand spawns a messenger thread with ⟦**send**(c, v)⟧ = **fork**($\lambda c'. c\ (c', v)$), which initializes the new thread with the term ($\lambda c'. c\ (c', v)$) ⟨k⟩ where ⟨k⟩ is the newly created barrier. The messenger thread then performs the β-reduction, resulting in an extra step on the ⅄ side.

To get a full operational correspondence (Gorla, 2010), we need a second "reflection" lemma stating that if the image of the translation ⟦ρ⟧ takes a step, then this step can be matched with a corresponding step in the GV semantics. Note that this only holds for well-typed terms: if we have the ill-typed term **receive**($\lambda x. x$) in the GV source, this is translated to well-typed ($\lambda x. x$)() in ⅄. Thus, whereas **receive**($\lambda x. x$) gets stuck in the GV semantics, its translation does not get stuck in the ⅄ semantics. We do expect a full operational correspondence to hold for well-typed terms, but while we have mechanized the proof of the simulation direction (Theorem 4.4.1), we have not mechanized a full operational correspondence. With a full operational correspondence it would be possible to lift ⅄'s deadlock freedom result to GV, and it would be interesting to see whether using ⅄ as a "proof IR" in this manner is a viable strategy for proving deadlock freedom of GV.

4.4.2 *Summary*

To add GV's session types to linear λ-calculus, we need to add the 5 new session type formers, the notion of duality, and the 7 session type operations. In contrast, ⅄ only adds one new operation, **fork**, and no new type formers and no notion of duality. Nevertheless, we have seen that we can encode session types in ⅄.

The encoding creates a new thread to store each message. Thus, ⅄ should not be viewed as a practical way to implement session types, but as a theoretical calculus, like other calculi that create one thread per message, such as the asynchronous π-calculus.

4.5 DEADLOCK FREEDOM, LEAK FREEDOM, AND GLOBAL PROGRESS

Linear typing in ⅄ guarantees strong properties for well-typed programs:

TYPE SAFETY: threads never get stuck, except by synchronizing with a barrier.

GLOBAL PROGRESS: a non-empty configuration can always take a step.

STRONG DEADLOCK FREEDOM: no subset of the threads gets stuck by waiting for each other.

MEMORY LEAK FREEDOM: all barriers in the configuration remain referenced by a thread.

These properties are all inequivalent in strength: none of the 4 properties is strictly stronger than another. In Section 4.5.3 we consider a property that is strictly stronger than these 4, but we will first focus on *global progress* (Section 4.5.1), as ideas behind the proof of global progress (Section 4.5.2), are also sufficient to prove the stronger property (Section 4.5.3).

4.5.1 *Global progress*

Let us consider the formal statement of global progress:

Theorem 4.5.1 (Global progress)**.**

$\quad$ *If* $\emptyset \vdash e : \mathbf{1}$*, and* $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho$*, then either* $\rho = \{\}$ *or* $\exists \rho'. \rho \rightsquigarrow \rho'$*.*

Intuitively, global progress states that if we start with a well-typed program, then any configuration we reach is either empty (*i.e.,* all threads have terminated and all barriers have been deallocated), or the configuration can perform a step. This property relies on linear typing, as ⅄ programs that violate linearity can deadlock and create a non-empty configuration that cannot step. A simple example is if one side does not use its barrier:

$$\textbf{let } x = \textbf{fork}(\lambda x'. \, ()) \textbf{ in } x \; 0 \quad \textcolor{red}{\text{Deadlock!}}$$

This deadlock is prevented by the linear type system, which ensures that each barrier is used *exactly once*. More complicated deadlocks are also possible in untyped programs, in which there is a circle of threads T_1 T_2 T_3 T_4 connected by barriers ● ● ● ● that are trying to synchronize (→▸—) in a cycle:
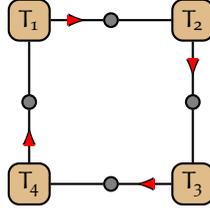
No thread can make progress because the threads are all synchronizing on different barriers. This shows that a simplistic scheme to prove deadlock freedom cannot work: we must somehow rule out such cycles. Fortunately, the linear type system ensures that the graph of connections between threads has the shape of a *forest* (*i.e.,* collection of trees, *i.e.,* an acyclic graph), and thus such circular deadlocks cannot happen. To see why the graph remains acyclic, consider what happens when we **fork**:

$$\textbf{let } x_1 = \textbf{fork}(...)$$
$$\textbf{let } x_2 = \textbf{fork}(...)$$
$$\textbf{let } x_3 = \textbf{fork}(...)$$
$$\textbf{let } y = \textbf{fork}(\lambda y'. \cdots x_2 \cdots x_3 \cdots)$$
$$\cdots x_1 \cdots$$

At the fourth **fork**, the barriers x_2 and x_3 are transferred to the new thread via lexical scoping, while the main thread keeps x_1 for itself. This is what happens to the graph:
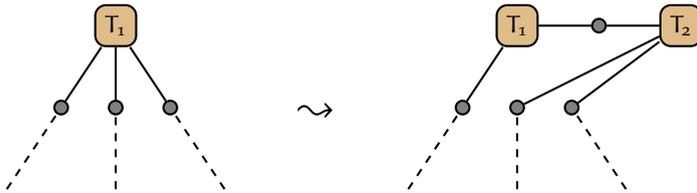
On the left hand side, we have the thread T_1 that is about to perform the fourth **fork**. It currently owns 3 barriers x_1, x_2, x_3, which are connected to the rest of the graph. After the fork, we have the new thread T_2, which is connected to T_1 by means of a new barrier. Crucially, the barriers x_2 and x_3 of the barriers that T_1 used to own were transferred to T_2 by means of lexical scoping. Nevertheless, as one can see in the figure above, if the graph of the configuration before the fork was acyclic, then the graph of the configuration after the fork is also acyclic. The same applies

to a synchronization step, when values containing potentially multiple barriers are exchanged:
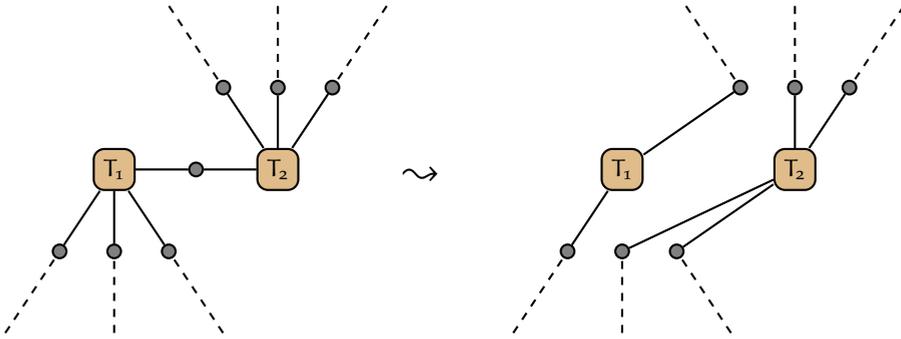
On the left, T_1 and T_2 each own 3 barriers, and they are also connected by a barrier. On the right, after the synchronization has taken place, the barrier between them has disappeared. Two of the barriers of T_1 were transferred to T_2, and one of the barriers of T_2 was transferred to T_1. Once again, if the graph on the left was acyclic, then the graph on the right will still be acyclic.[6]

The other operations of λ do not change the connections in the graph. Therefore, a program starts with a single thread, and then grows and alters its graph in a dance of fork and sync steps, but the graph remains acyclic at all times.

In the next section we will see in a bit more detail how the acyclicity of the graph is used in the proof of global progress.

RELATED WORK   Graph theoretic deadlock-freedom arguments are common in the session types literature and have previously been made by Carbone (Carbone and Debois, 2010), Lindley and Morris (Lindley and Morris, 2015), and Fowler, Kokke, Dardha, Lindley, and Morris (Fowler et al., 2021).

4.5.2 *Structure of the global progress proof*

This section gives more detail about how the acyclicity of the connection graph is used to prove global progress. For the full details, the interested reader is referred to the mechanization (Section 4.7).

Global progress states that if the configuration is non-empty, then it can take a step. Formally, there are several types of stuck configurations to rule out. Perhaps the configuration has a single thread and no barriers, but the thread is stuck on a type error (violating type safety). Or perhaps the configuration consists of just one barrier and no threads (violating memory leak freedom). Or perhaps there is a single

---

6 When one looks at the picture of the synchronizing threads, it seems that T_1 becomes totally disconnected from T_2 after the synchronization. This means that the threads can only communicate *once*. Yet the session-typed channel encoding seems to make it possible to communicate several times. Exercise for the reader: what is the solution to this paradox?

$$\frac{\cdot}{k:\tau;\emptyset \vdash \langle k \rangle : \tau} \qquad \frac{\cdot}{\emptyset;x:\tau \vdash x : \tau} \qquad \frac{\Sigma;\Gamma,x:\tau_1 \vdash e : \tau_2}{\Sigma;\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}$$

$$\frac{\Sigma_1;\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Sigma_2;\Gamma_2 \vdash e_2 : \tau_1}{\Sigma_1,\Sigma_2;\Gamma_1,\Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

Figure 28: Run-time type system (selected rules).

thread and a single barrier, and the thread is trying to synchronize but is stuck due to the absence of a partner to synchronize with (violating deadlock freedom).

The aforementioned stuck configurations and more complicated variations are ruled out by keeping track of sufficient type information and local invariants for each object in the configuration (*e.g.,* that a barrier is always connected to exactly two threads, and that the types of the references to the barrier are dual $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$). The interesting case is when the types are all locally correct, but the configuration is still deadlocked due to cyclic waiting. This case is ruled out by proving that well-typed programs maintain the invariant that the connection graph is acyclic, which implies that cyclic waiting cannot happen.

Formally, we define a *well-formedness* invariant of configurations, that maintains well-typedness of the threads as well as the acyclicity of the connection graph. For the well-typedness, we use the run-time type system in Figure 28.[7] The rules of the run-time type system correspond to the rules of the static type system, plus one additional rule for typing barrier literals $\langle k \rangle$. The typing judgment uses an additional Σ-context for typing those barrier literals.

We say that a configuration ρ is *well-formed* if we have an *acyclic* graph G (*i.e.,* an undirected forest) where the vertices correspond to the entries of the configuration, and the edges (between threads and barriers) are labeled with types, such that for each vertex i in G:

- If $\rho(i) = \text{Thread}(e)$ then $\Sigma;\emptyset \vdash e : \mathbf{1}$, where Σ is given by the types on the edges of the barriers connected to vertex i.

- If $\rho(i) = \text{Barrier}$ then vertex i has edges to two different threads, labeled with dual types $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$.

These conditions ensure that the graph structure matches the structure of the configuration: if we have a barrier literal $\langle k \rangle$ somewhere in the expression e of thread n, then the first condition ensures that we have an edge between n and k, labeled with a type $\tau_1 \multimap \tau_2$ to make expression e well-typed. The second condition then ensures that there is a second thread with barrier literal $\langle k \rangle$ in its expression, with

---

[7] Our run-time type system in Coq makes use of separation logic, following (Jacobs et al., 2022b). This is equivalent to the type system in the figure, but easier to work with in a proof assistant.

type $\tau_2 \multimap \tau_1$. Note that the two occurrences of the very same barrier literal $\langle k \rangle$ have two different types in the two different threads.

Now that the configuration invariant has been defined, proof of global progress (Theorem 4.5.1) can be structured as follows. Using well-typedness, we are able to show that the only way a configuration can get stuck is if all threads are trying to synchronize with a barrier. The invariant maintains that the graph structure is always acyclic. By a mathematical argument about graphs and the pigeonhole principle, we are then able to show that two of the threads must be synchronizing on the same barrier. Hence, at least one synchronization step can proceed, and we have global progress.

4.5.3 *Strengthened deadlock and memory leak freedom*

Global progress (Theorem 4.5.1) rules out whole-program deadlocks. It also ensures that all barriers have been used when the program finishes. However, this theorem does not guarantee anything as long as there is still a single thread that can step. Thus it does not guarantee local deadlock freedom, nor memory leak freedom while the program is still running, and it does not even guarantee type safety: a situation in which a thread is stuck on a type error is formally not ruled out by this theorem as long as there is another thread that can still step.

Our goal is to find a formulation that is strictly stronger than these 4 properties, and from which they can be easily proved as corollaries. We take inspiration from (Jacobs et al., 2022b), and find a strengthened formulation of deadlock freedom on the one hand, and strengthened memory leak freedom on the other hand. These strengthened formulations of deadlock freedom and memory leak freedom are equivalent to each other, and they imply type safety and global progress. In order to state these, we need the relation i waiting$_\rho$ j, which says that $i \in \text{dom}(\rho)$ is waiting for $j \in \text{dom}(\rho)$.

Intuitively, the meaning of waiting$_\rho$ is as follows. When a barrier k gets allocated, the literal $\langle k \rangle$ appears in two threads n_1, n_2. In this case we say that k waiting$_\rho$ n_1 and k waiting$_\rho$ n_2, because the barrier is waiting until the threads want to synchronize with it. Note that this relationship is dynamic: if the barrier literal $\langle k \rangle$ is transferred to another thread, then the barrier is waiting for that new thread to synchronize with it. Whenever a thread n starts trying to synchronize with k by calling $\langle k \rangle$ v for some value v, then the waiting relationship flips: we now say that the thread is waiting for the barrier, *i.e.*, that n waiting$_\rho$ k. Formally:

Definition 4.5.2. ⚙ We have i waiting$_\rho$ j if either:

1. $\rho(i) = \text{Barrier}$ and $\rho(j) = \text{Thread}(e)$ and $\langle i \rangle \in e$, but $e \neq K[\ \langle i \rangle\ v\ ]$, or

2. $\rho(i) = \text{Thread}(e)$ and $\rho(j) = \text{Barrier}$, and $e = K[\ \langle j \rangle\ v\ ]$

Using this notion, we can define what a partial deadlock/leak is. Intuitively, a partial deadlock is a situation in which there is some subset of the threads that are

all waiting for each other. Because our notion of waiting also incorporates barriers, we generalize this to say that a *partial deadlock/leak* is a situation in which there is some subset of the threads and barriers that are all waiting for each other. Formally:

Definition 4.5.3 (Partial deadlock/leak ⚙). Given a configuration ρ, a non-empty subset $S \subseteq \text{dom}(\rho)$ is in a partial deadlock/leak if these two conditions hold:

1. No $i \in S$ can step, *i.e.*, for all $i \in S$, $\neg \exists \rho'. \rho \overset{i}{\leadsto} \rho'$

2. If $i \in S$ and i waiting$_\rho$ j then $j \in S$

This notion also incorporates memory leaks: if there is some barrier that is not referenced by a thread, then the singleton set of that barrier is a partial deadlock/leak. Similarly, a single thread that is not synchronizing on a barrier, is considered to be in a singleton deadlock if it cannot step. This way, the notion of partial deadlock incorporates type safety.

Definition 4.5.4 (Partial deadlock/leak freedom ⚙). A configuration ρ is deadlock/leak free if no $S \subseteq \text{dom}(\rho)$ is in a partial deadlock/leak.

We also strengthen the standard notion of memory leak freedom, namely reachability, to incorporate aspects of deadlock freedom.

Definition 4.5.5 (Reachability ⚙). We inductively define the threads and barriers that are *reachable* in ρ: $j_0 \in \mathbb{N}$ is reachable in ρ if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geqslant 0$) such that j_0 waiting$_\rho$ j_1, and j_1 waiting$_\rho$ j_2, ..., and j_{k-1} waiting$_\rho$ j_k, and finally j_k can step in ρ, *i.e.*, $\exists \rho'. \rho \overset{j_k}{\leadsto} \rho'$.

Intuitively, an element $j_0 \in \mathbb{N}$ is reachable if j_0 can itself step or has a transitive waiting dependency on some j_k that can step. This notion is stronger than the usual notion of reachability, which considers objects to be reachable even if they are only reachable from threads that are blocked.

Definition 4.5.6. ⚙ A configuration ρ is *fully reachable* if all $i \in \text{dom}(\rho)$ are reachable in ρ.

As in (Jacobs et al., 2022b), our strengthened formulations of deadlock freedom and full reachability are equivalent for λ:

Theorem 4.5.7. ⚙ *A configuration ρ is deadlock/leak free if and only if it is fully reachable.*

Furthermore, these notions imply global progress and type safety:

Definition 4.5.8. ⚙ A configuration ρ has progress if $\rho = \emptyset$ or $\exists \rho', i. \rho \overset{i}{\leadsto} \rho'$.

Definition 4.5.9. ⚙ A configuration ρ is safe if for all $i \in \text{dom}(\rho)$, either $\exists \rho', i. \rho \overset{i}{\leadsto} \rho'$, or $\exists j. i$ waiting$_\rho$ j.

Theorem 4.5.10. ⚙ ⚙

If a configuration ρ is deadlock/leak free (or equivalently, fully reachable), then ρ has the progress and safety properties.

Our main theorem is thus that configurations that arise from well-typed programs are fully reachable and deadlock free:

Theorem 4.5.11. ⚙ *If $\emptyset \vdash e : \mathbf{1}$ and $\left\{ o \mapsto \mathsf{Thread}(e) \right\} \rightsquigarrow^{*} \rho'$, then ρ' is fully reachable and deadlock/leak free.*

The proof of the reachability half of Theorem 4.5.11 proceeds similarly to the proof of global progress (Theorem 4.5.1). The difference between the proofs is that the proof of reachability needs to explicitly keep track of the reason why each object in the configuration is reachable, whereas global progress only needs to find one of the "roots" of the reachability relation (*i.e.,* threads that can step).

Theorem 4.5.7 can then be used to obtain the deadlock freedom side of Theorem 4.5.11. The idea of the proof of Theorem 4.5.7 is that the set of all unreachable objects forms a deadlock, if the set is non-empty.

4.6 EXTENDING λ WITH UNRESTRICTED AND RECURSIVE TYPES

We add unrestricted types and recursive types as extensions. These can be omitted for a minimalistic language, but they enable us to do ordinary functional programming and recursive sessions in λ, bringing it closer to a realistic language in terms of features and expressiveness. The extended set of types is:

$$\tau \in \mathsf{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau \mid \tau \rightarrow \tau \mid \mu a.\tau \mid a$$

An equi-recursive interpretation of $\mu x.\tau$ avoids explicit (un)fold constructs (Crary et al., 1999). Recursive types make the language Turing complete, because they allow us to define recursive functions with the Y-combinator[8], and together with sums and products can be used to encode algebraic data types. Because session types are encoded as ordinary types in λ, recursive sessions are automatically supported. This includes recursion through the message, as in $\mu s.\ \mathbf{?}s.\mathsf{End}$, which is encoded as $\mu a.\ \mathbf{1} \multimap (\mathbf{1} \multimap \mathbf{1}) \times a$.

Formally and in the mechanization (Section 4.7), we use the coinductive method of Gay, Thiemann, and Vasconcelos (Gay et al., 2020) to handle equi-recursive types. This means that we formally do not have a syntactic $\mu a.\tau$ type constructor; instead we let the language of types be *coinductively generated*. Intuitively, this means that infinite types are allowed, and a recursive type $\mu a.\mathsf{F}(a)$ is represented as its infinite unfolding $\mathsf{F}(\mathsf{F}(\mathsf{F}(\cdots)))$. We use a meta-level fixpoint (CoFixpoint in Coq) to construct infinite/circular types. By using this method we do not need an additional typing rule for unfolding recursive types, since types are already identified up to unfolding.

---

8 One could also add an explicit **letrec**.

In order to make interesting use of recursive types, it is necessary to add *unrestricted types*, which are types for which the linearity restriction is lifted, so that they can be duplicated and discarded freely. A linear function can only be called once and hence cannot call itself, so recursive functions must have unrestricted type. Using unrestricted and recursive types, we do not need built-in recursion as we can encode recursive functions using the Y-combinator:

$$Y : ((\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)) \rightarrow (\tau_1 \rightarrow \tau_2)$$
$$Y \triangleq \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Here too it is apparent that since x is used twice, unrestricted as well as recursive types are required to type check the Y-combinator (Jacobs et al., 2022b).

In particular, we add the unrestricted function type $\tau_1 \rightarrow \tau_2$ as a new type former. We can then define rules that determine which of the existing types are unrestricted, as follows:

- $0, 1$ are unrestricted types

- $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are unrestricted if τ_1 and τ_2 are unrestricted

- $\tau_1 \rightarrow \tau_2$ is always unrestricted, even for linear τ_1 and τ_2

- $\tau_1 \multimap \tau_2$ is always linear, even for unrestricted τ_1 and τ_2

Using unrestricted function types, we can encode the !A connective from linear logic as $1 \rightarrow A$, and ?A as $A \rightarrow 1$.

Formally and in the mechanization, unrestricted types are handled by splitting the typing context using a 3-part relation split $\Gamma\ \Gamma_1\ \Gamma_2$, which intuitively says that $\Gamma \equiv \Gamma_1, \Gamma_2$, where variables of unrestricted type in Γ may occur in both Γ_1 and Γ_2 (see (Jacobs et al., 2022b) for details). We also make use of the predicate Γ unr to indicate that all variables in Γ must have unrestricted type. To extend the typing rules in Figure 26, we use split to split the context, and we use Γ unr wherever an empty context is required in Figure 26. For example, here are the typing rules for variables and for pairs:

$$\frac{\Gamma \text{ unr}}{\Gamma, x:\tau \vdash x : \tau} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2 \qquad \text{split } \Gamma\ \Gamma_1\ \Gamma_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

The other rules and the rules of the run-time type system are amended analogously.

Unrestricted and recursive types are purely type-level features and require no extensions to the expression language or to the operational semantics. Nevertheless, they upgrade λ to a Turing complete functional and concurrent language. For more details, we refer the interested reader to the mechanization (Section 4.7).

All our theorems have been mechanized in Coq. We use the connectivity graph library of (Jacobs et al., 2022b) in our mechanization. The mechanization is built-up as follows:

- Language definition: expressions, static type system, and operational semantics. Our mechanization includes the extensions with unrestricted and recursive types.

- A run-time type system that extends the static type system to barrier literals ⟨k⟩. The run-time type system is expressed in separation logic.

- A configuration well-formedness invariant, stating that the configuration remains well-typed, and the connectivity between threads and barriers remains acyclic.

- Proof that well-formedness is maintained by the operational semantics.

- Proof that well-formed configurations have the *full-reachability* property.

- Proofs that full-reachability is equivalent to deadlock/leak freedom, and that they imply type safety and global progress.

- The definition of the encoding of session types in λ, and proofs that the usual session typing rules are admissible.

- A definition of GV and its operational semantics, and the translation into λ, with a proof that the GV semantics can be simulated with the λ semantics. A lock-step simulation is obtained by inserting extra no-op steps in the GV semantics wherever λ does an administrative β-reduction.

Whereas the mechanized deadlock freedom proof for GV's session types by (Jacobs et al., 2022b) consists of 2139 lines of Coq definitions and proofs (excluding (Jacobs et al., 2022b)'s graph library), our mechanization of λ and its deadlock freedom is only 1229 lines. The encoding of GV's session types into λ together with the proofs of admissibility of the typing rules is 249 lines, and the operational simulation result is 309 lines.

Although the λ mechanization relies on connectivity graphs (Jacobs et al., 2022b), the techniques presented there were not immediately sufficient for proving deadlock freedom of λ. The difficulty lies in λ's sync step in the operational semantics, which exchanges resources between two vertices that are not directly adjacent in the graph. This is not supported as an operation by (Jacobs et al., 2022b), so we instead want to separate it into multiple smaller graph transformations. Unfortunately, the intermediate states do not satisfy the configuration invariant. The solution to this was to add extra "ghost state" to the labels on the edges of the graph, which keeps track of which sub-step of the decomposed graph transformation the connected vertices are in. As part of future work, it would be interesting to investigate whether this technique can be used more generally for composing graph transformations on the separation logic level, when the intermediate states do not satisfy the invariant.

4.8 RELATED WORK

Session types were originally described by Honda (Honda, 1993), and later by Honda, Vasconcelos, and Kubo (Honda et al., 1998). Gay and Vasconcelos (Gay and Vasconcelos, 2010) embedded session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus (Gay and Vasconcelos, 2010) was not yet deadlock free, Wadler's subsequent GV (Wadler, 2012) and its derivatives (Lindley and Morris, 2015, 2016c, 2017; Fowler et al., 2019, 2021) were.

Wadler also described the relation of GV to classical processes (CP) (Wadler, 2012), giving a kind of Curry-Howard correspondence between session types and classical linear logic. For intuitionistic linear logic, such a correspondence had earlier been described by Caires and Pfenning (Caires and Pfenning, 2010).

Lindley and Morris (Lindley and Morris, 2016b,c) give a CPS encoding of GV. The target of the CPS encoding is linear λ-calculus *without* fork, which is even more minimal than ⅄. The key difference between the CPS encoding and our encoding into ⅄ is that the CPS encoding is *global* (*i.e.,* a whole-program transformation), whereas the encoding of sessions into ⅄ is *local*, which is made possible by the built-in concurrency of ⅄. In other words, in contrast to the CPS encoding, the encoding of channel operations in ⅄ can be viewed as syntactic abbreviations or macros, satisfying Felleisen's expressiveness criterion (Felleisen, 1991).

When session types are added to the syntax of standard π-calculus they give rise to additional separate syntactic categories, which leads to a duplication of effort in the theory. Kobayashi showed that session types are encodable into standard π-types (Kobayashi, 2002b). Dardha, Giachino, and Sangiorgi formalize and extend Kobayashi's approach (Dardha et al., 2012, 2017). The encoding makes use of the fact that the π-calculus semantics has communication in the form of π-channels, and can thus encode session communication into π-communication. The encoding of multi-step sessions into single-shot π-channels sends a continuation along, so that the communication can continue. ⅄'s encoding of session types takes inspiration from this work, and also sends along continuations on which the communication can continue. On the other hand, ⅄ starts with λ-calculus, which does not have any concurrency or communication. We therefore *add* concurrency and communication to linear λ-calculus in the form of fork. Unlike the π-calculus' channel communication, which is one-way (like an individual step of a session), ⅄'s barrier communication atomically *exchanges* two values, so that barriers may be given linear function type A ⊸ B. This lets ⅄ get away with not adding any new type formers to linear λ-calculus, by reusing the quintessential λ-calculus type (the function type) for its communication primitive.

Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries, for instance by Scalas and Yoshida (Scalas and Yoshida, 2016a), Padovani (Padovani, 2017), and Kokke and Dardha (Kokke and Dardha, 2021a).

More distantly, Arslanagic, Pérez, and Voogd developed *minimal session types* (Arslanagic et al., 2019), which decompose multi-step session types into single-step "minimal" session types of the form **!**τ.End and **?**τ.End in a π-calculus. Whereas ⅄ and the preceding approaches (Kobayashi, 2002b; Dardha et al., 2012, 2017) encode sequencing by nesting payload types, minimal session types "slice" the n actions of a session s into indexed names $s_1, \ldots, s_n$, each having a minimal session type. Correct sequencing is arranged on the process level with additional synchronizations, using Parrow's decomposition of processes into trios (Parrow, 1998).

Niehren, Schwinghammer and Smolka developed a concurrent λ-calculus with futures (Niehren et al., 2005). Futures are akin to mutable variables that can only be assigned once. If a future has not been assigned a value yet, then attempting to read its value will block until a value becomes available. In addition to a non-linear type system which allows run-time errors due to multiple assignments to the same future, the authors also present a linear type system that ensures that futures are not assigned twice. The authors are able to build channels on top of futures by starting with the ordinary linked list data type, and making the tail of the list a future, thus making the list open-ended. Unlike session-typed channels, which follow a protocol and can send values of different types at different points in the protocol, their channels always communicate values of the same type. Besides the difference between futures (which are unidirectional) and ⅄'s barriers (which are bidirectional), another difference is that deadlock freedom is not guaranteed for all well-typed programs.

Aschieri, Ciabattoni and Genco give a Curry-Howard correspondence for Gödel Logic (Aschieri et al., 2017), which is intuitionistic logic extended with the axiom $(A \to B) \lor (B \to A)$[9]. This is a classical axiom that is implied by, but strictly weaker than the law of the excluded middle, and Gödel Logic thus provides an intermediate between intuitionistic and classical logic. The idea for the Curry-Howard interpretation of the axiom is that two copies of the continuation can be run in parallel, and exchange their evidence for A and B if both sides try to apply the implication obtained from the axiom. An important difference with ⅄ is that Gödel Logic is based on intuitionistic logic (corresponding to the ordinary simply typed lambda calculus), whereas ⅄ is based on the linear simply typed lambda calculus, and strongly relies on linearity for type safety and deadlock freedom. It would be interesting to investigate whether a connection between ⅄ and Gödel Logic can be established, but a naive attempt at interpreting the axiom as $(A \to B) + (B \to A)$ in ⅄ appears bound to fail, because in a linear setting the continuation/context cannot be duplicated without breaking the meta-theoretical properties.

We base our mechanization on the *connectivity graph* approach of Jacobs, Balzer, and Krebbers (Jacobs et al., 2022b), and we use their library to reason about graphs in Coq. This is related to the graphical approach of Carbone (Carbone and Debois, 2010), the proof method of Lindley and Morris (Lindley and Morris, 2015), and to the

---

9 Thanks to Dan Frumin for pointing out this connection.

abstract process structures of Fowler, Kokke, Dardha, Lindley, and Morris (Fowler et al., 2021).

More distantly, λ is inspired by minimal languages such as MiniJava (Roberts, 2001), MiniML (Myreen and Owens, 2012), the DOT calculus (Amin et al., 2016), and others.

4.9 CONCLUDING REMARKS

We have investigated λ, a minimal linear lambda calculus extended with **fork** to make it concurrent. We have seen that channel operations and linear session types can be encoded in λ, and we have shown that the resulting semantics for the channel operations simulates the GV semantics.

The metatheory of λ, including strong deadlock freedom, has been mechanized in Coq. Because of λ's minimality, the proofs are simpler and shorter than earlier mechanized proofs for session types. I hope you enjoyed this approach to distilling session types into a simple core, and hope that λ may serve as a minimal basis upon which future work may build.

Part II

SEPARATION LOGICS FOR MESSAGE PASSING

Chapter 5

Dependent Session Protocols in Separation Logic from First Principles

ABSTRACT   We develop an account of dependent session protocols in concurrent separation logic for a functional language with message-passing. Inspired by minimalistic session calculi, we present a layered design: starting from mutable references, we build one-shot channels, session channels, and imperative channels. Whereas previous work on dependent session protocols in concurrent separation logic required advanced mechanisms such as recursive domain equations and higher-order ghost state, we only require the most basic mechanisms to verify that our one-shot channels satisfy one-shot protocols, and subsequently treat their specification as a black box on top of which we define dependent session protocols. This has a number of advantages in terms of simplicity, elegance, and flexibility: support for subprotocols and guarded recursion automatically transfers from the one-shot protocols to the dependent session protocols, and we easily obtain various forms of channel closing. Because the meta theory of our results is so simple, we are able to give all definitions as part of this chapter, and mechanize all our results using the Iris framework in less than 1000 lines of Coq.

5.1   INTRODUCTION

Message passing is a commonly used abstraction for concurrent programming, with languages such as Erlang and Go having native support for it, and languages such as Java, Scala, Rust, and C# having library support. Session types offer powerful type systems for message passing concurrency (Honda, 1993; Honda et al., 1998), and have been extended with a number of exciting features:

1. **Dependent protocols:** The key ingredient of a session type system is the notion of a *session protocol*, which describes what data should be exchanged. For example, the session protocol **!Z.!Z.?B.end** expresses that two integers are sent, after which a Boolean is received, and the channel is closed. In vanilla session types, protocols were meant to specify the *types* of the exchanged data. They cannot be used to express that the right *values* are exchanged (*i.e.,* functional correctness), nor to express data-dependent protocols where the remaining protocol can depend on prior messages.

   There have been two lines of work to extend session protocols with logical conditions to remedy this shortcoming. Bocchi et al. (2010); Toninho et al.

(2011); Zhou et al. (2020); Thiemann and Vasconcelos (2020) develop type systems that combine concepts from the theory of dependent and refinement types with session types. Lozes and Villard (2012); Craciun et al. (2015); Hinrichsen et al. (2020) develop program logics that combine concurrent separation logic (O'Hearn, 2004; Brookes, 2004) with concepts from session types. Separation logic (instead of a type system) is used to enforce affine use of a channel library, and Hoare triple specifications (instead of typing rules) are provided for channel operations.

2. **Integration in functional languages:** While session types were originally developed in the context of π-calculus, a tempting direction is to combine session types with functional programming. In such languages, session-typed channels are considered first-class data, and can be stored in data types and sent over channels (similar to first-class mutable references in ML). The GV family by Gay and Vasconcelos (2010); Wadler (2012) extends linear lambda-calculus with channels. The SILL family by Toninho et al. (2013); Pfenning and Griffith (2015); Toninho (2015) uses a monadic embedding of session types into an unrestricted language.

3. **Session channels as a library:** Session types are typically a language feature, but a recent trend is to embed channels with session types as a library in an existing language (Hu et al., 2008; Scalas and Yoshida, 2016a; Pucella and Tov, 2008). Often, either the host language or the encoding supports substructural types, to enforce the affine use of session channels (Kokke and Dardha, 2021b; Lindley and Morris, 2016a; Jespersen et al., 2015b; Chen et al., 2022).

4. **Minimalistic calculi:** Session-typed languages add a large number of additional constructs to the types and expressions of their base languages. Already in the early days of session types, Kobayashi (2002b) showed that session types can be encoded into π-types; an approach formalized by Dardha et al. (2012, 2017), and applied to GV-style languages by Jacobs et al. (2022a).

5. **Mechanization:** The meta theory of session types is notorious for its complexity. There exist various published broken proofs—including the failure of subject reduction for several multiparty systems (Scalas and Yoshida, 2019). As a result, over the last 5 years there has been an extensive amount of work on the mechanization of session types by among others Thiemann (2019); Rouvoet et al. (2020); Hinrichsen et al. (2020, 2021); Tassarotti et al. (2017); Goto et al. (2016); Ciccone and Padovani (2020); Castro-Perez et al. (2020); Gay et al. (2020); Jacobs et al. (2022b); Castro-Perez et al. (2021).

To our knowledge, there is no prior work that combines all five features under a single roof. The goal of this functional pearl is thus to do exactly that. We will develop an account of dependent session protocols for a GV-style language in a concurrent separation logic. We start from *first principles*, enabling us to take a minimalistic

approach. Our results have been mechanized in the Coq proof assistant using the Iris framework for concurrent separation logic (Jung et al., 2015, 2016; Krebbers et al., 2017a; Jung et al., 2018b; Krebbers et al., 2018, 2017b). In the remainder of the introduction, we give a teaser of our approach and list some of our key insights.

KEY IDEA #1: IMPLICIT BUFFERS THROUGH ONE-SHOT CHANNELS   The first step to formalizing a language with message-passing concurrency is to decide on the semantics of channels. A common approach is to use an asynchronous semantics where the sender enqueues the messages in a buffer, from which the receiver dequeues them. In such a semantics, the receive operation can block if no message is present, but the send operation will always succeed immediately. To model the notion of a buffer, one typically incorporates a linked list in the formal definition of the language, and extends the language with operations to send (enqueue) and receive (dequeue) messages.

To be minimalistic, we want to avoid having to explicitly model the notion of a linked list in our semantics. Inspired by Kobayashi (2002b); Dardha et al. (2017); Jacobs et al. (2022a) we build on top of one-shot channels. These come with functions **new1** (), which creates a new channel; **send1** $c\,v$, which send a message v on channel c (without blocking); and **recv1** c, which receives a message v from c (blocks until a message has been sent). On top of the one-shot channels, we define regular multi-shot session channels. For example, the send operation of session channels is defined as:

$$\textbf{send}\,c\,v \triangleq \textbf{let}\,c' = \textbf{new1}\,()\,\textbf{in}\,\textbf{send1}\,c\,(v, c');\ c'$$

This operation not only sends the message v, but also creates a new channel c' for the remainder of the communication, and sends the new channel paired with the message. While there is no explicit notion of a buffer or linked-list in the semantics of one-shot channels, nor in the definition of session channels, we will show that the buffer arises implicitly from the preceding definition.

KEY IDEA #2: DEPENDENT SESSION PROTOCOLS VIA ONE-SHOT PROTOCOLS
Program logics for message-passing concurrency typically come with a *channel points-to* connective $c \rightarrowtail p$, which provides unique ownership of a channel endpoint c that has to obey to a protocol p. These protocols typically have a sequenced structure, describing a dependent session of multiple exchanges. An example of a *dependent separation protocol* in the Actris logic by Hinrichsen et al. (2020, 2022) is $!(n : \mathbb{N})\langle n\rangle.\,!(m : \mathbb{N})\langle m\rangle\{n \leqslant m\}.\,?\langle m - n\rangle.\,\textbf{end}$. This protocol expresses that two natural numbers $n \leqslant m$ are sent, and the difference $m - n$ is returned.

Similar to our desire for avoiding the need to explicitly model the buffers that underpin channels as linked lists, we would like to avoid having to inductively define such dependent session protocols. In our system, the channel points-to connective for the one-shot channels is simply $c \rightarrowtail (tag, \Phi)$, where $tag \in \{\textsf{Send}, \textsf{Recv}\}$ and Φ is a predicate over the exchanged value. While our protocols only describe a
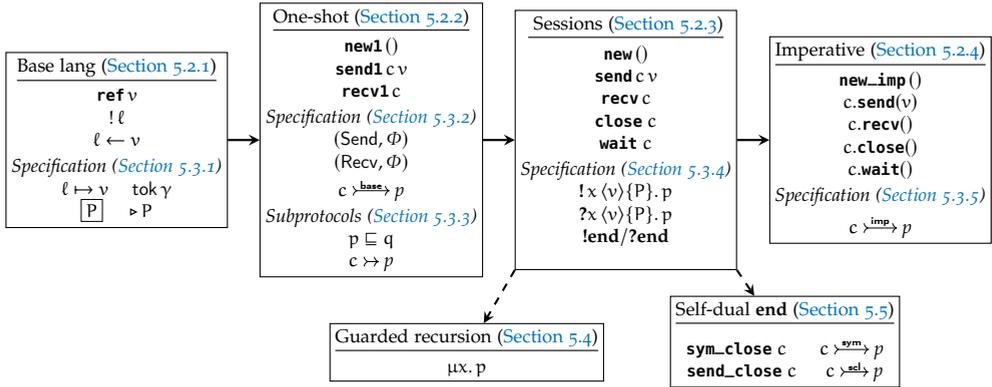
Figure 29: Layered design of our development.

single message, dependent session protocols that can describe session channels are simply defined as combinators. This is achieved by recursively using the channel points-to connective for describing the channel continuation inside the base protocol Φ. Due to Iris's support for impredicativity (Svendsen and Birkedal, 2014), we can use its fixpoint combinator to define recursive (and dependent) protocols by guarded recursion.

KEY IDEA #3: LAYERED SESSION CHANNEL LIBRARY DESIGN AND VERIFICATION We implement session channels in terms of one-shot channels, and our dependent session protocols as combinators of one-shot protocols, but we wish to go further by layering our design—from below and above. The layered design is shown in Figure 29.

From below, we do not start with a language that has channels as primitive. We build on top of a functional language with mutable references as found in languages of the ML family (with allocation, deallocation, store and load). One-shot channels are implemented on top of primitive mutable references, and verified using (Iris's) separation logic rules for the verification of concurrent programs with mutable shared-memory references. Building on top of a language with mutable references has other tangible benefits. First, we can write and verify programs that transfer data by reference. Second, we can define both functional versions of session channels (that return a new endpoint) and imperative versions of channel endpoints (that mutate the channel).

From above, we demonstrate the flexibility of our solution by implementing multiple methods for closing a session. Session types and protocols are often terminated with an explicit **end**-tag, and it is non-trivial to extend the range of termination tags in settings where the protocols are defined inductively. Since our session channels are defined as combinators on top of the one-shot channels—that do not inherently include a method for closing—we can freely choose how to close our channels, after the fact. Initially, we implement asymmetric closing, where one

endpoint initiates the closing of a channel (protocol **!end**), while the other waits and actually deallocates the memory backing the channel (protocol **?end**). We later provide two alternatives with a self-dual **end** protocol: symmetrically closing the channel with a the same closing operation on both endpoints, where the last call deallocated the channel, and a combined send-close operation, which sends a last message but does not create a continuation channel.

KEY IDEA #4: MECHANIZATION USING A SUBSET OF IRIS   Our layered design proved beneficial for the meta theory and mechanization of our results. We only need the usual points-to connective $\ell \mapsto v$ for ownership of locations ℓ with value v in separation logic, a simple form of ghost state (unique tokens), and Iris's impredicative invariants. By comparison, the Actris logic by Hinrichsen et al. (2020, 2022) relies on Iris invariants, as well as a non-trivial model of recursive protocols using the technique from America and Rutten (1989) for solving recursive domain equations, and uses Iris's mechanism for higher-order ghost state (Jung et al., 2016) to define its channel points-to connective $c \rightarrowtail p$. Our approach still critically relies on Iris' invariants, but we do not need to solve a custom recursive domain equation, nor do we need to define custom higher-order ghost state. Iris' invariants are constructed via a recursive domain equation and higher-order ghost state under the hood, and our approach re-uses this standard machinery as a black box, and we thereby avoid the need to interact with recursive domain equations and higher-order ghost state directly. Since the meta theory of our results is so simple, we are able to give all definitions as part of this chapter (there is no appendix) and mechanize all our results in less than 1000 lines of Coq.

CONTRIBUTIONS   This chapter makes the following contributions:

- A layered implementation of higher-order shared-memory session channels, starting from mutable references, on which we build one-shot channels, session channels, and imperative channels (Section 5.2)

- A layered development of separation logic specifications for our channels. We start from a small subset of Iris, developing specifications for one-shot channels, which are then treated as a black box upon which we build high-level dependent separation protocols (Section 5.3)

- Support for subprotocols (Section 5.3.3) and guarded recursion (Section 5.4), which transfers *automatically* from one-shot protocols to dependent session protocols.

- A demonstration of the extensibility obtained by building on first principles, through various methods for closing session channels (Section 5.5)

- A small and intuitive mechanization in the Coq proof assistant, comprised of less than 1000 lines of Coq code (Section 5.7). The paper is annotated with

mechanization icons (⚙) that link to the relevant Coq code, and a cross-reference sheet is provided (§ A).

5.2 LAYERED IMPLEMENTATION OF CHANNELS

In this section we will implement message passing channels in terms of low-level operations. We build these channels in several layers:

- We start by describing the base language and its low-level operations (Section 5.2.1).

- We then build a library of one-shot channels (Section 5.2.2).

- On top of this we build functional multi-shot session channels (Section 5.2.3).

- As a final layer, we have imperative session channels (Section 5.2.4).

- We show that linked lists (buffers) implicitly emerge (Section 5.2.5).

In the subsequent Section 5.3, we develop specifications and proof for each of the layers, and demonstrate how to verify the correctness of the example.

5.2.1 *Base Language*

We use HeapLang, a low-level concurrent language that comes with the Iris separation logic framework, as our base language. HeapLang has the purely functional operations that one would expect, such as arithmetic and conditionals, and also includes products and sums. For the purpose of this chapter, the following operations on mutable memory locations are the most relevant:

ref v	Allocate a new memory location that initially stores value v.
$!\ell$	Read the value from memory location ℓ.
$\ell \leftarrow v$	Write value v to location ℓ.
free ℓ	Free the memory location ℓ.

HeapLang additionally includes a primitive for spawning a new thread:

fork $\{e\}$ Run program e in a new thread.

The program e is allowed to refer to variables in the surrounding lexical context. The following is a grammar of the most notable constructs that we will use:

$$e \in \mathsf{Expr} ::= \mathbf{ref}\, e \mid\, !\, e \mid e \leftarrow e \mid \mathbf{fork}\ \{e\} \mid \mathbf{free}\, e \mid$$
$$\mathbf{Some}\, e \mid \mathbf{None} \mid \mathbf{match}\, e\, \mathbf{with}\, \mathbf{Some}\, x \Rightarrow e;\, \mathbf{None} \Rightarrow e\, \mathbf{end} \mid$$
$$x \mid e\, e \mid \lambda x.e \mid \mathbf{assert}(e) \mid \mathbf{for}(x = e..e)\, e \mid \ldots$$

5.2.2 *One-Shot Channels*

At the base of our development lie one-shot channels, which communicate a single message from a sender to a receiver. The API consists of the following operations:

new1()   Create and return a new one-shot channel c.

send1 $c\,v$ Send message v on channel c (non-blocking).

recv1 c Receive message v from channel c (blocks until a message is sent).

The channels are one-shot; only one value is sent over the channel, after which point the channel is deallocated as a part of **recv1** c.

EXAMPLE OF USING ONE-SHOT CHANNELS   These channels enable us to set up a communication between child and parent threads as in the following example:

$$\text{prog\_single} \triangleq \qquad\qquad\qquad\qquad\qquad\qquad ⚙$$

> **let** $c = $ **new1**() **in**
> **fork** $\{$**let** $l = $ **ref** 42 **in send1** $c\,l\}$;
> **assert**(!(**recv1** c) $= 42$)

The main thread creates a one-shot channel c, which is shared between the main thread and a forked-off thread. The forked-off thread then dynamically allocates a reference to 42, and sends the location over the channel. Finally, the main thread receives the reference, reads it, and asserts that the stored value is 42. To communicate several times, we could share several channels, but an interesting alternative style that allows unbounded communication is to send a new channel along with the message, as we shall see in Section 5.2.3.

In the HeapLang semantics, **assert** gets stuck if the condition is false. Safety (the fact that the **assert** does not fail) crucially depends on the forked-off thread not modifying the reference after it has sent it. This example is safe as the exclusive permission to write and read the reference first belongs to the forked-off thread, after which it is transferred to the main thread. We verify this safe transfer of ownership in Section 5.3.2. This goes beyond standard session types due to reference ownership and the verification of the **assert**.

IMPLEMENTATION OF ONE-SHOT CHANNELS   In our development, channels are not primitive but implemented in terms of low-level mutable references. A channel is represented as a mutable reference that initially contains the value **None**. To send a value v to the channel, we set the mutable reference to **Some** v. To receive from the channel, we read the value of the mutable reference in a loop, until we see the **None**

change to **Some** v. We then deallocate the mutable reference, and return v. This gives us the following implementation:

$$\mathbf{new1}\,() \triangleq \mathbf{ref}\ \mathbf{None}$$ ⚙

$$\mathbf{send1}\,c\,v \triangleq c \leftarrow \mathbf{Some}\,v$$ ⚙

$$\mathbf{recv1}\,c \triangleq \mathbf{match}\,!\,c\ \mathbf{with}$$ ⚙
$$|\ \mathbf{Some}\,v \Rightarrow \mathbf{free}\,c;\ v$$
$$|\ \mathbf{None}\ \ \Rightarrow \mathbf{recv1}\,c$$
$$\mathbf{end}$$

This implementation shows that safety also depends on the fact that clients only call **recv1** once, and does not call **send1** after a completed **recv1**. These would otherwise result in a double-free and use-after-free, which get stuck in the HeapLang semantics.

HeapLang has a sequentially consistent memory model. In a weaker memory model, the store/load instructions should use release/acquire memory order options (or stronger). Similar to most literature on Iris—with the exception of papers specifically focused on weak memory (Mével and Jourdan, 2021; Kaiser et al., 2017; Dang et al., 2020)—we ignore these concerns.

5.2.3 *Session Channels*

A session channel facilitates sequences of messages between two channel endpoints, which is useful for implementing client-server style concurrency. Session channels have the following API:

new ()   Create a new session channel.

send $c\,e$ Send message e on channel c, and return a continuation channel.

recv c Receive a pair (v, c') of the message v and continuation channel c'.

close c Send termination message.

wait c Wait for the termination message and deallocate the channel.

In this section we demonstrate how one-shot channels can be used to implement session channels. The session channels are obtained by allocating and exchanging a new one-shot channel whenever a value is sent. The new one-shot channel is then

used as a continuation of the session. The session channels are implemented as follows:

$$\mathbf{new}\,() \triangleq \mathbf{new1}\,() \qquad \text{⚙}$$
$$\mathbf{send}\,c\,v \triangleq \mathbf{let}\,c' = \mathbf{new1}\,()\,\mathbf{in}\,\mathbf{send1}\,c\,(v, c');\; c' \qquad \text{⚙}$$
$$\mathbf{recv}\,c \triangleq \mathbf{recv1}\,c \qquad \text{⚙}$$
$$\mathbf{close}\,c \triangleq \mathbf{send1}\,c\,() \qquad \text{⚙}$$
$$\mathbf{wait}\,c \triangleq \mathbf{recv1}\,c \qquad \text{⚙}$$

The **new** function allocates an initial one-shot channel and returns it as the session channel. The **send** function allocates a new one-shot channel, and sends it along the original channel with the given message v, after which the new channel is returned. The **recv** function receives the value and continuation channel pair using the original one-shot channel receive function. The **close** function sends a final termination flag, without allocating a new one-shot channel, to terminate the session. The **wait** function receives the final termination flag, which deallocates the channel.

For session channels to be used safely—*i.e.,* to not cause memory errors such as use-after-free or double-free—it is crucial that channel endpoints are used in a dual way. That is, if there is a **send** on one endpoint, there should be a matching receive on the other endpoint, and *vice versa*. Similarly, a **close** should match up with a **wait**. We discuss other options for closing channels in Section 5.5.

EXAMPLE OF USING SESSION CHANNELS   An example of using the session channels is as follows:

```
prog_add ≜                                                        ⚙
  let c = new () in
  fork {let (l, c) = recv c in l ← (!l + 2); let c = send c () in wait c};
  let l = ref 40 in
  let c = send c l in let (_, c) = recv c in close c;
  assert(!l = 42)
```

Here, the main thread initially creates a session channel c, which is shared between the main thread and forked-off 'worker' thread. The main thread dynamically allocates a reference to 40, after which it sends the reference over the channel. The worker thread receives the reference, adds 2 to it, and sends a flag back, to signal that the reference has been updated. The main thread receives the flag and then reads the updated value stored in the reference, and asserts that it is 42. Finally, the main thread sends the closing signal, which is received by the worker thread. Each operation on the channel binds the channel continuation to an overshadowing name c, to intuitively capture that they keep working on the same session.

Similar to the example presented in Section 5.2.1, this program is safe if the **assert** succeeds and there are no memory errors due to improper use of the channel API. Intuitively, this example achieves safe access to the reference l via ownership delegation over the channel. We verify this in Section 5.3.4.

5.2.4 *Imperative Channels*

Although session channels are more convenient to use than one-shot channels, they still require us to continuously pass around new channel references. On top of session channels we therefore define *imperative* channels, which have a traditional imperative channel API:

new_imp()	Create a new imperative channel, and return a pair of endpoints.
c.**send**(v)	Send message v on channel c. Return nothing.
c.**recv**()	Receive a message from channel c. Return only the message.
c.**close**()	Send termination message and close the channel.
c.**wait**()	Wait for termination message and close the channel.

We implement imperative channels in terms of session channels by storing a session channel in a mutable reference:

$$\textbf{new\_imp}\,() \triangleq \textbf{let}\,c = \textbf{new}\,()\,\textbf{in}\,(\textbf{ref}\,c, \textbf{ref}\,c) \qquad \text{✿}$$

$$c.\textbf{send}(v) \triangleq c \leftarrow \textbf{send}\,(!\,c)\,v \qquad \text{✿}$$

$$c.\textbf{recv}() \triangleq \textbf{let}\,(v, c') = \textbf{recv}\,!\,c\,\textbf{in}\,c \leftarrow c';\,v \qquad \text{✿}$$

$$c.\textbf{close}() \triangleq \textbf{close}\,(!\,c);\,\textbf{free}\,c \qquad \text{✿}$$

$$c.\textbf{wait}() \triangleq \textbf{wait}\,(!\,c);\,\textbf{free}\,c \qquad \text{✿}$$

5.2.5 *Emerging Linked List Buffers*

We demonstrate the imperative API with the example from Figure 30. The example creates a channel to communicate between the main thread and the forked-off 'worker' thread. The main thread allocates a reference s and sends the message $(100, s)$ to the worker thread, which indicates that the main thread is going to send 100 further number messages to the worker thread. The worker thread receives each of these numbers, and mutates s to keep track of their sum. Finally, the worker thread sends an empty acknowledgment message () to the main thread, indicating that it is done with s and will not mutate s further. The main thread closes the session by sending the closing signal, which the worker thread waits for. The main thread then reads the value of the sum from s, and asserts that it is correctly computed.

The linked structures that emerge during execution are displayed in Figure 31. In the picture, the main thread has sent the numbers $[1, \ldots, 9]$, while the worker thread

```
let (c₁, c₂) = new_imp () in          — create channel between the threads ⚙
fork {                                — start the worker thread
  let (n, s) = c₂.recv() in           — receive count n and answer reference s
  for(i = 1..n) s ← c₂.recv() + ! s   — sum n received numbers
  c₂.send(())                         — signal that we are done
  c₂.wait()                           — wait for closing signal
}
let s = ref(0) in                     — mutable reference to store the sum
c₁.send((100, s))                     — we send 100 numbers to be summed into s
for(i = 1..100) c₁.send(i)            — send the numbers 1..100
c₁.recv()                             — wait until the worker is done
c₁.close()                            — send closing signal
assert(! s == 5050)                   — assert that the received answer is correct
```

Figure 30: An example program using the imperative channels.
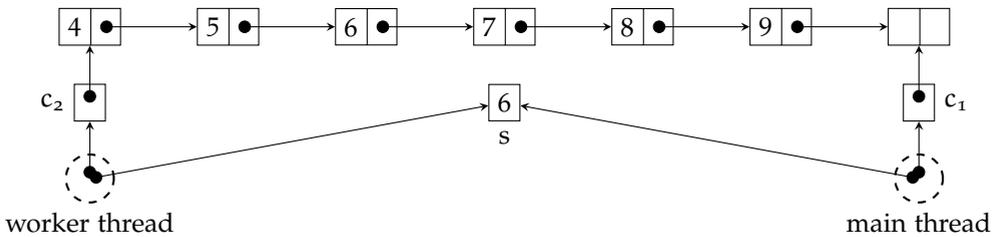
Figure 31: The heap structure emerging from the example in Figure 30, after the first 9 values $[1, \ldots, 9]$ have been sent, and the first 3 $[1, 2, 3]$ have been received and summed in the shared location s. The boxes with a number n and next pointer ℓ indicate that the memory location contains **Some**(n, ℓ), and the empty box on the right indicates that the memory location contains **None**.

has so far only received $[1, 2, 3]$. At run time, the worker thread will have a reference to c_2, which points to the head of a linked list structure. When the worker thread receives the next message (4), it updates c_2 to point to the next linked list element, and adds the value of the message to s. The main thread also has a reference to s, but it will not use it until the worker thread has sent the completion signal back, to avoid race conditions. Instead, the main thread is still busy working on the other end of the linked list. Each time the main thread sends a message, it allocates a new memory location, puts its message into the tail, and updates the tail of the existing linked list to point to the new location. This emergence of the linked list occurs because the send operation allocates a new one-shot channel, represented as a memory location, and sends it along with the message. At a lower level of abstraction, this results in a linked list buffer of messages, where each message is a pair of a value and a continuation channel.

If the worker thread were to catch up with the main thread, it would wait until it sees a message. When the main thread is done, it tries to *receive* a message using the last linked list node it has created, which is initially still empty. When the client reaches that node, it puts the acknowledgment () into it, signaling that the main thread may now read from s.[1] More generally, the threads switch roles when the polarity of the protocol changes: the thread that used to consume list cells now creates new list cells, and *vice versa*.

Note that the emergence of the buffer as a bi-directional linked list is somewhat implicit. We have built several layers of channels, but at no point did we have to think about the linked-list run-time structure as a whole. We will see a similar phenomenon when doing the proofs: we never need to think about the run-time structure as a whole. Instead, we will develop specifications in a layered way, following the layers of the implementation.

In the remainder of this chapter, we will develop specifications for these different layers (corresponding to Sections 5.2.1 to 5.2.4), and prove the correctness of the channel implementations with respect to these specifications. We can then use the specifications to verify this example in Section 5.3.5.

5.3 LAYERED SPECIFICATIONS AND VERIFICATION

As the reader may have noticed, the implementations in the preceding section are *untyped*. Rather than assigning types to the channel APIs, we will provide *separation logic specifications*. These allow us to prove functional correctness of programs that make use of the channel API. We prove *partial correctness*, which guarantees that if a program satisfies a separation logic specification with trivial precondition, then the program is safe, *i.e.,* does not get stuck in the semantics due to run-time type errors, use-after-free or double-free bugs, or failing **assert** expressions. In terms of session types, our result should be compared with *type safety* and *session fidelity*. As

---

1 In Section 5.5.2 we will see a different way of closing the channel, which does not require this acknowledgment.

is standard in papers that use Iris, we do not prove deadlock freedom or termination (which would only be true when assuming a fair scheduler as the spin-loop in `recv1` could otherwise trivially loop).

In this section we first present the Iris separation logic that we use to verify our implementation (Section 5.3.1). We then show how we verified the one-shot channel implementation using Iris primitives (Section 5.3.2), and layer subprotocols on top of it (Section 5.3.3). We verify dependent separation protocol (Hinrichsen et al., 2020, 2022) specifications of our session channel implementation directly on top of our one-shot specifications (Section 5.3.4). Finally, we verify our imperative channel implementation in terms of the session channel specifications (Section 5.3.5).

5.3.1 *The Iris Separation Logic*

To specify and verify the channel implementations and example clients, we use the Iris separation logic. Figure 32 shows the grammar and a selection of rules of the subset of Iris that we use. Iris provides a program logic for HeapLang with Hoare-triples $\{P\}\, e\, \{\varPhi\}$, which express that given the precondition (P : iProp), the program (e : Expr) is safe to execute, and yields the postcondition ($\varPhi$: Val $\rightarrow$ iProp). We often write $\{P\}\, e\, \{w.\, Q\} \triangleq \{P\}\, e\, \{\lambda w.\, Q\}$ and $\{P\}\, e\, \{Q\} \triangleq \{P\}\, e\, \{\lambda w.\, w = ()\, \ast\, Q\}$.

Iris is a separation logic (O'Hearn et al., 2001), meaning that propositions assert ownership over resources, such as references. This is made precise by the separation logic connectives, such as the separating conjunction P $\ast$ Q, which describes that the propositions P and Q holds for *separate* parts of the heap. In particular, this lets us derive *exclusivity* of references; it is impossible to separately own the same reference: $\ell \mapsto v \ast \ell \mapsto w \mathbin{-\!\ast} \mathsf{False}$. Here $\mathbin{-\!\ast}$ is the "separating implication" connective. It acts similarly to the regular implication, but for separation logic.

Separation logic facilitates *modular verification*, by virtue of the framing rule Ht-FRAME, which states that we can verify programs e in the presence of separate resources R. Non-structured concurrency is supported by the Ht-FORK rule. Finally, Iris enjoys the conventional rules for mutable references Ht-ALLOC, Ht-LOAD, Ht-STORE, and Ht-FREE, which respectively allow allocating, reading, updating, and freeing mutable references.

We use Iris's impredicative invariants $\boxed{P}$, ghost state tokens $\mathsf{tok}\,\gamma$, and later modality $\triangleright P$. We further discuss the meaning and importance of these connectives throughout the section.

5.3.2 *One-Shot Channels*

In Figure 33 we show separation logic specifications for the one-shot channel implementation from Section 5.2.2. These specifications make use of *one-shot protocols* that describe the protocol for a one-shot channel. As a one-shot channel communicates a value, the protocol will carry a predicate describing which values

Iris propositions:

$$P, Q \in iProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid \qquad \text{(Propositional logic)}$$
$$\forall x.\, P \mid \exists x.\, P \mid x = y \mid \qquad \text{(Higher-order logic with equality)}$$
$$P * Q \mid P \twoheadrightarrow Q \mid \ell \mapsto v \mid \{P\}\, e\, \{\Phi\} \mid \qquad \text{(Separation logic)}$$
$$\boxed{P} \mid \text{tok}\,\gamma \mid \triangleright P \mid \ldots \qquad \text{(Invariants, ghost state, and step indexing)}$$

Separation logic:

HT-FRAME
$$\frac{\{P\}\, e\, \{w.\, Q\}}{\{P * R\}\, e\, \{w.\, Q * R\}}$$

HT-VAL
$$\{\text{True}\}\, v\, \{w.\, w = v\}$$

HT-FORK
$$\frac{\{P\}\, e\, \{\text{True}\}}{\{P\}\, \textbf{fork}\, \{e\}\, \{\text{True}\}}$$

Heap manipulation:

HT-ALLOC
$$\{\text{True}\}\, \textbf{ref}\, v\, \{\ell.\, \ell \mapsto v\}$$

HT-LOAD
$$\{\ell \mapsto v\}\, !\,\ell\, \{w.\, (w = v) * \ell \mapsto v\}$$

HT-STORE
$$\{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\}$$

HT-FREE
$$\{\ell \mapsto v\}\, \textbf{free}\, \ell\, \{\text{True}\}$$

Invariants*, ghost state, and step indexing:

HT-INV-ALLOC
$$\frac{\{\boxed{P} * Q\}\, e\, \{\Phi\}}{\{\triangleright P * Q\}\, e\, \{\Phi\}}$$

HT-INV-OPEN-CLOSE
$$\frac{e \text{ is atomic} \qquad \{\triangleright P * Q\}\, e\, \{w.\, \triangleright P * R\}}{\{\boxed{P} * Q\}\, e\, \{w.\, R\}}$$

HT-LATER-FRAME
$$\frac{e \text{ is not a value} \qquad \{P\}\, e\, \{w.\, Q\}}{\{P * \triangleright R\}\, e\, \{w.\, Q * R\}}$$

HT-LATER-TIMELESS
$$\frac{R \text{ is timeless} \qquad \{P * R\}\, e\, \{w.\, Q\}}{\{P * \triangleright R\}\, e\, \{w.\, Q\}}$$

HT-GHOST-ALLOC
$$\frac{\{P * \exists \gamma.\, \text{tok}\,\gamma\}\, e\, \{\Phi\}}{\{P\}\, e\, \{\Phi\}}$$

TOK-EXCL
$$\frac{\text{tok}\,\gamma * \text{tok}\,\gamma}{\text{False}}{}^{*}$$

LÖB
$$\frac{\triangleright P \twoheadrightarrow P}{P}\Box$$

Figure 32: The grammar and a selection of rules of Iris.
*Iris uses *masks* to prevent opening the same invariant twice during a single step, as that is unsound (Jung et al., 2018b). We omit details about this mechanism because we only open at most one invariant at every step.

Protocols:	$p \in \text{Prot} \triangleq \{\text{Send}, \text{Recv}\} \times (\text{Val} \to \text{iProp})$	⚙
Dual:	$\overline{(\text{Send}, \Phi)} \triangleq (\text{Recv}, \Phi) \qquad \overline{(\text{Recv}, \Phi)} \triangleq (\text{Send}, \Phi)$	⚙
Points-to:	$c \rightarrowtail^{\textbf{base}} p \in \text{iProp}$ where $p \in \text{Prot}$ and $c \in \text{Val}$	⚙
New:	$\{\text{True}\}\ \textbf{new1}\,()\ \{c.\ c \rightarrowtail^{\textbf{base}} p * c \rightarrowtail^{\textbf{base}} \overline{p}\}$	⚙
Send:	$\{c \rightarrowtail^{\textbf{base}} (\text{Send}, \Phi) * \Phi\,v\}\ \textbf{send1}\,c\,v\ \{\text{True}\}$	⚙
Receive:	$\{c \rightarrowtail^{\textbf{base}} (\text{Recv}, \Phi)\}\ \textbf{recv1}\,c\ \{\Phi\}$	⚙

Figure 33: Separation logic specifications for one-shot channels.

are allowed to be communicated with that channel. Additionally, the protocol says whether we are allowed to send or receive. Therefore, we represent one-shot protocols as a pair $(\textit{tag}, \Phi)$ where $\textit{tag} \in \{\text{Send}, \text{Recv}\}$ and $\Phi \in \text{Val} \to \text{iProp}$. The predicate Φ is a separation logic predicate, so that protocols can express transfer of ownership.

To link protocols to actual channels, we shall define a *channel points-to* predicate $c \rightarrowtail^{\textbf{base}} (\textit{tag}, \Phi)$. The channel points-to provides unique ownership of one end of the channel and says that channel c satisfies protocol $(\textit{tag}, \Phi)$. The channel points-to is analogous to the normal points-to $\ell \mapsto v$ of separation logic, in the sense that a points-to assertion is required to verify an invocation of a channel operation. The definition can be found in Figure 34, but we will first discuss how it is used in the Hoare rules for the channel operations.

When we create a new channel using **new1**(), we may choose the protocol predicate Φ, and we get *two* channel points-tos: $c \rightarrowtail^{\textbf{base}} (\text{Send}, \Phi)$ and $c \rightarrowtail^{\textbf{base}} (\text{Recv}, \Phi)$. Note that we get both channel points-tos for the same channel c, because the same memory location is used for both ends of the channel, and the two channel points-tos represent ownership of the two ends of the channel, which give two different views of the same memory location. As we shall see in Section 5.3.2, this is achieved by moving the ownership of the primitive heap points-to of the memory location into an invariant, which allows us to share it. In accordance with session types, and to state the specification of **new1**() in a symmetric manner (Figure 33), we introduce the *dual* function on protocols, given by $\overline{(\text{Send}, \Phi)} \triangleq (\text{Recv}, \Phi)$ and $\overline{(\text{Recv}, \Phi)} \triangleq (\text{Send}, \Phi)$.

Once we have the two channel-point-to predicates we may give one of them to another thread, and keep one of them in the current thread. This way we ensure that two threads use the protocol to agree on how the channel will be used.

We may then use the **send1** and **recv1** operations to perform the communication. The **send1** $c\,v$ operation requires ownership of $c \rightarrowtail^{\textbf{base}} (\text{Send}, \Phi)$ as well as $\Phi\,v$ in its precondition. Dually, the **recv1** c operation requires ownership of $c \rightarrowtail^{\textbf{base}} (\text{Recv}, \Phi)$ in its precondition. Its postcondition guarantees that **recv1** c returns a value v that

satisfies Φv. With these specifications we can verify the example presented in Section 5.2.2 with the following protocol:

$$p_{single} \triangleq (\mathsf{Send}, \lambda(v : \mathsf{Val}). \exists(\ell : \mathsf{Addr}). v = \ell * \ell \mapsto 42) \qquad \text{⚙}$$

This protocol expresses that the exchanged value v is a location ℓ. We transfer the ownership of the exchanged reference ℓ along with the message. With this, we can symbolically apply the one-shot channel specifications, and finally assert that the value read from the received reference is 42.

VERIFYING THE IMPLEMENTATION WITH RESPECT TO THE SPECIFICATION
We now prove that the one-shot channel implementation satisfies its specification. To do this, we *define* the channel points-to $c \xrightarrow{\textbf{base}} p$ in terms of Iris logic primitives (namely, ordinary points-to, ghost state and invariants). We then prove that the specifications for **new1**, **send1** and **recv1** follow from the rules of Iris. We first present the two key concepts from Iris needed for our proof: *ghost state* and *invariants*.

GHOST STATE   Ghost state is logical state that we can use to logically coordinate between parallel threads. Compared to the standard approach to ghost state in concurrency verification (Owicki and Gries, 1976), ghost state in Iris is not part of the program text. It is introduced and manipulated solely in proofs. Just as the physical heap keeps track of the values of memory locations, Iris has a ghost heap that keeps track of the values of ghost locations. In our case we only need the very simplest form of ghost state: we need pure ownership over ghost heap locations; we do not need to store further information in the ghost locations. Given the ghost location γ, we have the ghost resource $\mathsf{tok}\,\gamma$, which is analogous to $\ell \mapsto ()$, *i.e.*, a location that points to a unit value. It may seem a bit puzzling that ghost locations that do not store any interesting contents can be helpful in a proof. The key is that ghost locations have the same *exclusivity* as memory locations. That is, we have the Tok-excl rule that says it is impossible to have ownership of two ghost locations with the same name: $\mathsf{tok}\,\gamma * \mathsf{tok}\,\gamma \wand \mathsf{False}$. We shall see why this is useful in a moment. Finally, we can always allocate new pieces of ghost state, using the Ht-ghost-alloc rule.

INVARIANTS   The points-to resource $\ell \mapsto v$ is an affine resource, and cannot be duplicated. This is a problem for verifying concurrent programs, where we would like to use the same memory location from multiple threads: when we fork off a child thread, we would like to keep ownership over the memory location in both the main thread and the child thread.

To solve this issue, concurrent separation logic has the notion of *invariants*. At any moment in the proof where we have ownership over $P \in \mathsf{iProp}$, we can choose to establish P as an invariant, denoted $\boxed{P} \in \mathsf{iProp}$. This is formally described by the Ht-inv-alloc rule. The advantage of an invariant is that it can be freely duplicated,

i.e., $\boxed{\text{P}} \twoheadrightarrow \boxed{\text{P}} * \boxed{\text{P}}$. In turn, we cannot directly access the P inside the invariant. Instead, we can only temporarily access it when the program takes an atomic step, such as a memory load $!\ell$ or store $\ell \leftarrow v$. After the atomic step has happened, we must immediately put P back into the invariant. This is formally described by the HT-INV-OPEN-CLOSE rule, where the resources $\triangleright$ P are the resources that are temporarily removed from the invariant. In the precondition of the rule, we obtain access to the resources $\triangleright$ P taken out of the invariant, and in the postcondition we have to give back the resources $\triangleright$ P, which represents putting them back into the invariant. The proposition P inside an invariant is typically a disjunction of several states, where the states may assert ownership over memory locations using $\ell \mapsto v$, and may assert that v has certain properties in that state. A state may also assert ownership over ghost resources.

Iris's invariants are *impredicative* (Svendsen and Birkedal, 2014), which effectively lets us nest invariants inside of invariants, because $\boxed{\text{P}} \in \mathsf{iProp}$ for every P $\in \mathsf{iProp}$, including P $= \boxed{\text{Q}}$. Nesting of invariants is critical for the verification of our session channels, as will be covered in Section 5.3.4. To maintain soundness of the Iris logic, resources P extracted from an invariant $\boxed{\text{P}}$ are guarded by a *later modality* $\triangleright$ P (Nakano, 2000; Appel et al., 2007). This later can be seen in the HT-INV-OPEN-CLOSE rule. Resources $\triangleright$ P behind a later modality can only be used after the program does the next step of execution. This is formally expressed by the HT-LATER-FRAME rule, which states that one can frame resources under a later, if the program has not terminated. Another means of stripping laters is if the guarded resources are *timeless* (HT-LATER-TIMELESS). Pure propositions, reference ownership ($\ell \mapsto v$) and ghost ownership (tok γ) are timeless, which means when we open an invariant, we can immediately remove the later from these connectives.

THE ONE-SHOT CHANNEL INVARIANT   To verify the one-shot channels, we need to define the connective $c \xrightarrow{\text{base}} p$, whose key ingredient is an invariant. To explain the invariant, we start with a key observation. The one-shot channel can be in three different states: (1) no message has been sent ($\ell \mapsto$ **None**), (2) a message has been sent but not received ($\ell \mapsto$ **Some** v), and (3) the message has been both sent and received (ℓ has been deallocated). These states are reflected in the invariant chan_inv γ_1 γ_2 ℓ Φ defined in Figure 34. The arguments γ_1 and γ_2 are two ghost locations, whereas ℓ is the physical memory location where the channel is located, and Φ is the predicate associated with the protocol. The invariant captures each state with a separate disjunct. By virtue of the exclusion of the ghost resources, it is then possible to exclude possible states, based on local ghost ownership. In particular, if one owns tok γ_1, the invariant must be in the first state (as the other states assert ownership of the token). Similarly, if one owns tok γ_2, the invariant cannot be in the final state. The proof then follows by letting the sender own tok γ_1 and the receiver own tok γ_2, to let them locally determine which state the invariant is in, by the exclusivity rule of the ghost resources.

More formally, with the invariant in place, we can define the channel points-to $c \rightarrowtail^{\textbf{base}} (tag, \Phi)$, as presented in Figure 34. The definition captures (1) that c is a reference (c = ℓ), (2) that the invariant is established ($\boxed{\text{chan\_inv } \gamma_1 \gamma_2 \ell \Phi}$), (3) that the endpoint has ownership of either $\text{tok } \gamma_1$ or $\text{tok } \gamma_2$, if they are the sender or receiver, respectively. The later modalities (▹) in the definition of $c \rightarrowtail^{\textbf{base}} p$ are needed to support infinite protocols via guarded recursion (Section 5.4).

Initially, when creating a channel, we establish the invariant in the first state, using the HT-INV-ALLOC rule. We then duplicate the invariant, and create $c \rightarrowtail^{\textbf{base}} (\text{Send}, \Phi)$ and $c \rightarrowtail^{\textbf{base}} (\text{Recv}, \Phi)$ using the two copies of the invariant, as well as $\text{tok } \gamma_1$ and $\text{tok } \gamma_2$, respectively, which are created by two applications of the HT-GHOST-ALLOC rule.

When the sender wants to send their message v, they temporarily open the invariant using the HT-INV-OPEN-CLOSE rule, and determine that they are in the first state, based on their $\text{tok } \gamma_1$ token. They then get ownership over the reference $\ell \mapsto \textbf{None}$. The sender then modifies the location to contain the sent value $\textbf{Some } v$, and transfers the ownership back into the invariant. The sender also puts the token $\text{tok } \gamma_1$ into the invariant, as well as the resources Φv captured by the protocol. The invariant is restored in the second state.

When the receiver wants to receive, it temporarily opens up the invariant, using the HT-INV-OPEN-CLOSE rule, to get ownership over the reference. It reads the location, and if the value is $\textbf{None}$, it determines that it is in the first state, and so it loops. Once a value $\textbf{Some } v$ is read, it is determined that we are in the second state, and so the receiver deallocates the reference. The receiver additionally takes the Φv resource out of the invariant, and re-establishes the invariant by putting its token $\text{tok } \gamma_2$ into the invariant, which restores it in the third state.

The rule for **new1** is then proven as follows. We obtain ownership over the location $\ell \mapsto \textbf{None}$ because **new1** allocates the reference. We also allocate two new ghost locations $\text{tok } \gamma_1$ and $\text{tok } \gamma_2$ obtaining the identifiers γ_1 and γ_2. We establish the invariant using the first disjunct, by putting $\ell \mapsto \textbf{None}$ into the invariant, and allocate it with the HT-INV-ALLOC rule. We then duplicate the invariant, and create $c \rightarrowtail^{\textbf{base}} (\text{Send}, \Phi)$ and $c \rightarrowtail^{\textbf{base}} (\text{Recv}, \Phi)$ using the two copies of the invariant, as well as $\text{tok } \gamma_1$ and $\text{tok } \gamma_2$, respectively.

5.3.3 *Subprotocols*

We define a subprotocol relation on dependent separation protocols as introduced by Actris (Hinrichsen et al., 2022), analogous to subtyping on session types (Gay and Hole, 2005). Whereas subtyping between session types is established by subtyping between the messages, the subprotocol relation between protocols is established by implications between the separation logic predicates.[2]

---

2 Similarly to asynchronous subtyping (Mostrous et al., 2009; Mostrous and Yoshida, 2015), the Actris subprotocols also enjoy *asynchronous subprotocolling*, which allow swapping sends in front of receives. Actris supports this due to its two-buffer semantics. As the semantics of our session channels, as built

$$\text{chan\_inv } \gamma_1 \, \gamma_2 \, \ell \, \Phi \triangleq (\underbrace{\ell \mapsto \textbf{None}}_{\text{(1) initial state}}) \vee (\underbrace{\exists v. \, \ell \mapsto \textbf{Some} \, v \ast \text{tok} \, \gamma_1 \ast \Phi \, v}_{\text{(2) message sent, but not yet received}}) \vee (\underbrace{\text{tok} \, \gamma_1 \ast \text{tok} \, \gamma_2}_{\text{(3) final state}})$$

⚙

$$c \xrightarrow{\textbf{base}} (tag, \Phi) \triangleq \exists \gamma_1, \gamma_2, \ell. \, \triangleright (c = \ell) \ast \boxed{\text{chan\_inv } \gamma_1 \, \gamma_2 \, \ell \, \Phi} \ast \triangleright \begin{cases} \text{tok} \, \gamma_1 & \text{if } tag = \text{Send} \\ \text{tok} \, \gamma_2 & \text{if } tag = \text{Recv} \end{cases}$$

⚙

Figure 34: The channel invariant and channel points-to definition.

The subprotocol relation is denoted $p \sqsubseteq q$ where p, q are protocols, and is defined as follows:

$$(tag_1, \Phi_1) \sqsubseteq (tag_2, \Phi_2) \triangleq \begin{cases} \forall v. \, \Phi_2 \, v \twoheadrightarrow \Phi_1 \, v & \text{if } tag_1 = tag_2 = \text{Send} \\ \forall v. \, \Phi_1 \, v \twoheadrightarrow \Phi_2 \, v & \text{if } tag_1 = tag_2 = \text{Recv} \\ \text{False} & \text{if } tag_1 \neq tag_2 \end{cases} \qquad ⚙$$

This relation is reflexive and transitive, and $p \sqsubseteq q$ iff $\overline{q} \sqsubseteq \overline{p}$. We layer subprotocols on top of our specification for one-shot channels by defining a new channel points-to $c \rightarrowtail p$ that is explicitly closed under subprotocols:

$$c \rightarrowtail p \triangleq \exists q. \, \triangleright (q \sqsubseteq p) \ast c \xrightarrow{\textbf{base}} q \qquad ⚙$$

We do not use a superscript on $c \rightarrowtail p$ because we consider it to be the main channel points-to, whereas we view $c \xrightarrow{\textbf{base}} q$ as an internal notion. This channel points-to satisfies a *subsumption* like rule: $(c \rightarrowtail p) \ast \triangleright (p \sqsubseteq q) \twoheadrightarrow (c \rightarrowtail q)$, which is proved by transitivity of $\sqsubseteq$. The use of the later modality ($\triangleright$) is discussed in [Section 5.4](). ⚙

We can prove versions of the specifications for **new1**, **send1**, and **recv1** for $\rightarrowtail$. These proofs are straightforward, because we can prove these specifications *using* the existing specifications for $\xrightarrow{\textbf{base}}$ from [Figure 33](), by using $p \sqsubseteq q$ at appropriate points to convert a $\Phi_1 \, v$ into $\Phi_2 \, v$ or *vice versa*. In particular, we apply this conversion in the send rule just before sending the message, and in the receive rule just after receiving the message. We also trivially have $(c \xrightarrow{\textbf{base}} p) \twoheadrightarrow (c \rightarrowtail p)$, which is used to prove the **new1** rule for $\rightarrowtail$. ⚙⚙⚙⚙

---

on single shot channels, corresponds to a single-buffer semantics, asynchronous subtyping is *unsound* in our setting. Our notion of subprotocols therefore focuses on the implication between separation logic predicates, and does not allow swapping sends in front of receives.

Protocols:	$(!\,x\,\langle v\rangle\{P\}.\,\mathsf{p} \mid ?x\,\langle v\rangle\{P\}.\,\mathsf{p} \mid !\mathbf{end} \mid ?\mathbf{end}) \in \mathsf{Prot}$	⚙
Dual:	$\overline{!\,x\,\langle v\rangle\{P\}.\,p} = ?x\,\langle v\rangle\{P\}.\,\overline{p}$ $\qquad$ $\overline{?x\,\langle v\rangle\{P\}.\,p} = !\,x\,\langle v\rangle\{P\}.\,\overline{p}$	⚙ ⚙
	$\overline{!\mathbf{end}} = ?\mathbf{end}$ $\qquad$ $\overline{?\mathbf{end}} = !\mathbf{end}$ $\qquad$ $\overline{\overline{p}} = p$	⚙ ⚙ ⚙
New:	$\{\mathsf{True}\}\ \mathbf{new}\,()\ \{c.\ c \rightarrowtail p * c \rightarrowtail \overline{p}\}$	⚙
Send:	$\{c \rightarrowtail (!\,x\,\langle v\rangle\{P\}.\,p) * P\ t\}\ \mathbf{send}\ c\ (v\ t)\ \{c'.\ c' \rightarrowtail (p\ t)\}$	⚙
Receive:	$\{c \rightarrowtail (?x\,\langle v\rangle\{P\}.\,p)\}\ \mathbf{recv}\ c\ \{w.\ \exists y, c'.\ w = (v\ y, c') * c' \rightarrowtail (p\ y) * P\ y\}$	
		⚙
Close:	$\{c \rightarrowtail !\mathbf{end}\}\ \mathbf{close}\ c\ \{\mathsf{True}\}$	⚙
Wait:	$\{c \rightarrowtail ?\mathbf{end}\}\ \mathbf{wait}\ c\ \{\mathsf{True}\}$	⚙

Figure 35: Dependent Separation Protocols and session channel specifications

5.3.4 *Session Channels*

Now that we have established the specifications for the one-shot channels, we move on to the next layer: multi-shot session channels. A prominent approach to specifying and verifying multi-shot channels is the concept of *session types* (Honda, 1993), which lets a user ascribe session channel endpoints with a sequence of obligations to send or receive messages of certain types. More recently, the session type approach has been adopted in the separation logic setting (Craciun et al., 2015; Hinrichsen et al., 2022). One such adaptation is *Dependent Separation Protocols* (Hinrichsen et al., 2022). Rather than ascribing types to each exchange, dependent separation protocols ascribe logical variables, physical values, and propositions. The dependent separation protocols and the specifications for the session channels can be seen in Figure 35.

The dependent separation protocols consists of four constructors: $!\,x\,\langle v\rangle\{P\}.\,p$, $?x\,\langle v\rangle\{P\}.\,p$, $!\mathbf{end}$, and $?\mathbf{end}$. The first two constructors describe the permission to send or receive the logical variable x, the value v, and the resources P, respectively, after which they follow the protocol tail p. Here, x binds into all of the remaining constituents. We often omit the binder when it is of the unit type: *e.g.,* $!\,\langle v\rangle\{P\}.\,p$. We similarly often omit the proposition if it is True: *e.g.,* $!\,x\,\langle v\rangle.\,p$. The last two constructors specify that the protocol has ended, meaning that no further operations can be made on the channel, and the channel can be closed. We further detail alternative specifications for closing and deallocation in Section 5.5.

The protocols are subject to the same notion of *duality*, as presented in Section 5.3.2. The dual of a protocol is the same sequence of obligations, where the polarity has been flipped, *i.e.,* all sends (**!**) become receives (**?**), and *vice versa*, as made precise by the rules of the figure. Finally, we use the same channel endpoint ownership $c \rightarrowtail p$

as for the one-shot channels, as the dependent separation protocols share the same type as the one-shot protocols, as will be seen momentarily.

The dependent separation protocols can be used to specify and verify session channels. As an example, the following dependent separation protocol specifies the interactions of the prog_add example from Section 5.2.3:

$$\text{prot\_add} \triangleq \, ! ((\ell, x) : \text{Addr} \times \mathbb{Z}) \, \langle \ell \rangle \{\ell \mapsto x\}. \, ? \langle () \rangle \{\ell \mapsto x + 2\}. \, \text{!end} \qquad \text{\ding{72}}$$

The protocol says that one must first send a reference to a number (captured by the logical variable (ℓ, x) : Addr $\times$ $\mathbb{Z}$)), along with the ownership of the reference $\ell \mapsto x$. Afterwards, the updated reference can be reacquired, followed by the protocol termination. The dual of the protocol is $?((\ell, x)$: Addr $\times$ $\mathbb{Z})\langle \ell \rangle \{\ell \mapsto x\}. ! \langle () \rangle \{\ell \mapsto x + 2\}. \, \text{?end}$.

The notion of duality is used in the specification for **new**. The specification states that we obtain separate exclusive ownership of the returned endpoint c, one with a freely picked protocol p and the other with its dual $\bar{p}$. This mimics the intuition from the one-shot channel, in which one endpoint had to release the specified resources, while the other could acquire them. The specification for **send** states that in order to send, the channel endpoint must have a sending protocol, and we must give up the specified resources P t, for a specific instantiation t of the variable x. Additionally, the sent value must correspond to the protocol, for the variable instantiation v t. As a result, the returned channel endpoint follows the protocol tail $c' \rightarrowtail p$ t, for the same variable instantiation. Conversely, the specification for **recv** states that we can receive if the channel endpoint has a receiving protocol. As a result we obtain an instance of the logical variable y, and the resources specified by the protocol Q y. Additionally, the returned value is exactly the one specified by the protocol v y, and the new endpoint follows the protocol tail $c' \rightarrowtail p$ y. The prog_add example can now be verified using the prot_add protocol. \ding{72}

VERIFICATION OF THE SESSION CHANNEL SPECIFICATIONS   The definitions of the dependent separation protocols and the specification rules presented in Figure 35 are derived directly on top of the one-shot channel definitions and specifications. In particular, the type of dependent separation protocols is the same as the one for the one-shot channel protocols, namely Prot. The definition of the receiving protocol is as follows:

$$\text{recv\_prot} \, (\tau : \text{Type}) \, (v : \tau \rightarrow \text{Val}) \, (P : \tau \rightarrow \text{iProp}) \, (p : \tau \rightarrow \text{Prot}) : \text{Prot} \triangleq \qquad \text{\ding{72}}$$
$$(Recv, \lambda(r : Val). \exists (x : \tau), (c : Val). \, r = (v \, x, c) * P \, x * c \rightarrowtail p \, x)$$

$$?(x : \tau) \, \langle v \rangle \{P\}. \, p \triangleq \text{recv\_prot} \, \tau \, (\lambda x. \, v) \, (\lambda x. \, P) \, (\lambda x. \, p)$$

The recv_prot constructor takes four arguments, and constructs a receiving one-shot channel protocol. In particular the constructor takes the type of its logical variable τ, the exchanged value v, the exchanged proposition P, and the protocol tail p. The

latter three arguments all abstract over the protocol variable, which is existentially quantified in the protocol body. The second projection captures that the actual exchanged value is a tuple of the value specified by the protocol (v x), and the continuation (c). It additionally includes ownership of the resources specified by the protocol (P x), and finally a one-shot channel ownership, of the continuation with the protocol tail ($c \rightarrowtail (p\ x)$). The notation $?(x : \tau)\langle v \rangle\{P\}.p$ then simply lets us instantiate the receiving constructor, without explicitly repeating the variable abstraction for the three constituents.

The duality function of the session channels is the same as the one for the one-shot channel. We define the sending constructor in terms of the receiving one, using the duality function as follows:

$$!x\,\langle v \rangle\{P\}.p \triangleq \overline{?x\,\langle v \rangle\{P\}.\overline{p}}$$

To specify the **close** and **wait** operations we define two session protocols:

$$?\mathbf{end} \triangleq (\mathsf{Recv}, \lambda r.\, r = ())$$

$$!\mathbf{end} \triangleq \overline{?\mathbf{end}}$$

Finally, the channel endpoint ownership $c \rightarrowtail p$ is identical to the one for the one-shot channels, as the type of the protocols are the same, they simply carry channel continuations now. This immediate reuse of the one-shot ownership is made possible by the higher-order nature of Iris. In particular, the internal invariant of the endpoint ownership refers to the session protocols, which internally includes a nested endpoint ownership, and so on. By virtue of the step-indexing of Iris, this is sound as we always take a step for each unfolding of the nested invariants.

With these definitions the soundness of the session channel specifications (Figure 35) follow almost immediately from the sound specifications of the one-shot channel operations **send1** and **recv1**.

SUBPROTOCOLS FOR SESSION PROTOCOLS   We have a notion of subprotocols for one-shot protocols (Section 5.3.3), but what about dependent session protocols? Because we have *defined* session protocols as particular forms of one-shot protocols, we get the appropriate notion of subprotocols for session protocols for free. The following lemmas for session subprotocols (and the imperative derivation on top of them) are *already true* and easily derived from the subprotocol rules in Section 5.3.3:

$$\frac{\forall x_1.\, \Phi_1\, x_1 \twoheadrightarrow \exists x_2.\, (v_1\, x_1 = v_2\, x_2) * \Phi_2\, x_2 * \triangleright(p_1\, x_1 \sqsubseteq p_2\, x_2)}{?x_1\,\langle v_1 \rangle\{\Phi_1\}.p_1 \sqsubseteq\, ?x_2\,\langle v_2 \rangle\{\Phi_2\}.p_2}$$

$$\frac{\forall x_2.\, \Phi_2\, x_2 \twoheadrightarrow \exists x_1.\, (v_2\, x_2 = v_1\, x_1) * \Phi_1\, x_1 * \triangleright(p_1\, x_1 \sqsubseteq p_2\, x_2)}{!x_1\,\langle v_1 \rangle\{\Phi_1\}.p_1 \sqsubseteq\, !x_2\,\langle v_2 \rangle\{\Phi_2\}.p_2}$$

At a high level, these lemmas state that a session protocol is a subprotocol of another, if for each logical message in the first protocol, there exists an appropriate logical message in the second protocol, such that we have a separating implication between separation logic assertions, and the tails of the protocols are in a subprotocol relationship. The stated lemmas are somewhat stronger than this high-level description; for instance, the user of the lemmas gets access to the assertion $\Phi_1 \, x_1$ *before* having to provide the corresponding logical message x_2 for the other protocol. As an example, this strengthening allows one to perform a form of *framing* of resources within a protocol: if a resource is provided by an earlier send and needed by a later receive, we can frame these two resources (*i.e.*, remove both from the protocol by canceling them out). This property can be illustrated by the following rules:

$$! x \, \langle v \rangle \{P\}. \, ? x \, \langle w \rangle \{Q\}. \, p \sqsubseteq \, ! x \, \langle v \rangle \{P * R\}. \, ? x \, \langle w \rangle \{Q * R\}. \, p \qquad \text{✿}$$

$$? x \, \langle v \rangle \{P * R\}. \, ! x \, \langle w \rangle \{Q * R\}. \, p \sqsubseteq \, ? x \, \langle v \rangle \{P\}. \, ! x \, \langle w \rangle \{Q\}. \, p \qquad \text{✿}$$

5.3.5 *Imperative Channels*

Because our session channels create new pointers at each step, they return new channels, and are thus inconvenient to work with. For that reason, we have our final layer: the imperative channels from Section 5.2.4. These channels put a session channel in a mutable reference, so that we can use the same mutable reference throughout and use mutating operations to change the reference to a new session channel upon send and receive operations. To handle these channels, we introduce a new channel points-to $c \rightarrowtail^{\mathbf{imp}} p$. The specifications for the imperative channels can be found in Figure 36. We note a couple of differences with respect to the session channels:

- The **new_imp** operation returns a pair of channels now, so the points-to connectives in the postcondition are for the two components of the pair.

- The **send** operation does not return a value. The new channel points-to in the postcondition refers to the original channel instead.

- The **recv** operation only returns one value—the message. The channel points-to in the postcondition once again refers to the original channel.

VERIFYING THE IMPERATIVE CHANNEL SPECIFICATIONS   To verify the session channels we first define a new connective for channel endpoint ownership:

$$c \rightarrowtail^{\mathbf{imp}} p \triangleq \exists (\ell : \mathsf{Addr}), (c' : \mathsf{Val}). \, c = \ell * \ell \mapsto c' * c' \rightarrowtail p \qquad \text{✿}$$

The new imperative channel ownership connective $c \rightarrowtail^{\mathbf{imp}} p$ simply lifts the original connective $c' \rightarrowtail p$ to assert ownership of a mutable reference.

Points-to: $c \xrightarrow{\text{imp}} p \in \text{iProp}$ where $p \in \text{Prot and } c \in \text{Val}$ ⚙

New: $\{\text{True}\} \text{ \textbf{new\_imp}}\,() \,\{w.\, \exists c_1, c_2.\, w = (c_1, c_2) * c_1 \xrightarrow{\text{imp}} p * c_2 \xrightarrow{\text{imp}} \overline{p}\}$ ⚙

Send: $\{c \xrightarrow{\text{imp}} (!\,x\,\langle v \rangle \{P\}.\,p) * P\,t\}\, c.\textbf{send}(v\,t)\,\{c \xrightarrow{\text{imp}} (p\,t)\}$ ⚙

Receive: $\{c \xrightarrow{\text{imp}} (?\,x\,\langle v \rangle \{P\}.\,p)\}\, c.\textbf{recv}()\,\{w.\, \exists y.\, w = v\,y * c \xrightarrow{\text{imp}} (p\,y) * P\,y\}$

⚙

Close: $\{c \xrightarrow{\text{imp}} \,!\textbf{end}\}\, c.\textbf{close}()\,\{\text{True}\}$ ⚙

Wait: $\{c \xrightarrow{\text{imp}} \,?\textbf{end}\}\, c.\textbf{wait}()\,\{\text{True}\}$ ⚙

Figure 36: Separation logic specifications for imperative channels.

With this definition in hand, verifying the specification is trivial. We simply use the Iris rule for allocating, reading, and updating the reference, along with the specifications for the original channel endpoint ownership, to resolve the operations on the channel.

Because the new channel-points-to is defined in terms of the old one, the results of subprotocols easily lift to the imperative channels. ⚙

VERIFYING THE EXAMPLE We now explain how these specifications can be used to verify the example from Figure 30. The example starts by allocating a new channel, so we use the specification for **new_imp**. In order to use this specification, we have to choose the session protocol p. We use the following protocol:

$$\text{prot\_sum}'\,x\,n \triangleq \textbf{if}\,n = 0\,\textbf{then}\,(?\langle() \rangle \{s \mapsto x\}.\,!\textbf{end})$$
$$\textbf{else}\,(!\,(y : \mathbb{N})\,\langle y \rangle.\,\text{prot\_sum}'\,(x + y)\,n)$$
$$\text{prot\_sum} \triangleq \,!\,((n, s) : \mathbb{N} \times \text{Addr})\,\langle (n, s) \rangle \{s \mapsto 0\}.\,\text{prot\_sum}'\,0\,n$$ ⚙

The protocol prot_sum says that we will first send the pair (n, s) of a number and a location, and the assertion that $s \mapsto 0$. We then continue with the protocol $\text{prot\_sum}'\,0\,n$, which is recursively defined. Its first argument keeps track of the sum of the messages sent so far, and the second argument keeps track of how many messages we still have to send. When the counter $n = 0$, we stop sending and instead receive a unit value, as well as the assertion that $s \mapsto x$, *i.e.,* the sum of the messages sent.

After the channel allocation, we have $c_1 \xrightarrow{\text{imp}} \text{prot\_sum}$ and $c_2 \xrightarrow{\text{imp}} \overline{\text{prot\_sum}}$. We verify the first interaction using the first step of prot_sum. We prove the loops correct using induction: the main thread does induction on 100, and the child thread induction on the received message n (which will be 100, but the child thread does not know this). After the final synchronization, the ownership over s has been

transferred back to the main thread. According to the protocol, the location s points to the value $1 + 2 + \cdots + 100$, which is equal to 5050 by mathematical reasoning. ⚙

As the reader can see, the reasoning about the pointer structure of the buffers is completely encapsulated in the higher-level session specifications. The nondeterminism present due to the asynchronous semantics of the send operation does not need to be reasoned about explicitly: although the depth of the linked list buffer changes non-deterministically according to the thread scheduling of the sends and receives, the proof does not explicitly reason about this at all.

5.4 GUARDED RECURSION

As we have seen in the example in Section 5.3.4, we can already create some recursive protocols by employing recursion over natural numbers (or other inductively-defined data types in Coq). Recursion over natural numbers lets us verify the example from Figure 30 where one side sends a number n, and then sends n further messages. Although recursion on inductive types is powerful, it does not allow us to create protocols for truly infinite interactions with services that run forever. We can create protocols that support truly infinite interactions with Iris's operator for *guarded recursion*.

Iris models guarded recursion via *step-indexing* (Appel and McAllester, 2001; Ahmed, 2004), meaning that separation logic propositions iProp are internally monotone predicates of a natural number i, the step index. Intuitively, the meaning of such a proposition is given by taking the limit to ever higher step indices. This allows us to model infinite protocols as a step-indexed protocol of unboundedly increasing depth. Iris does not expose the step index to the user of the logic, so we cannot define protocols by direct recursion over i. Instead, Iris provides a *logical* account of step-indexing (Appel et al., 2007; Dreyer et al., 2011) through the later modality $\triangleright P$ (Nakano, 2000), and a guarded recursion operator $\mu x.F\ x$ for constructing recursive predicates. The $F\ x$ must be *contractive* in the sense that recursive occurrences of x in F must only occur under a later $\triangleright$. This ensures that creating such a recursive predicate does not result in any logical paradoxes. Our protocols $\mathsf{Prot} \triangleq (\mathsf{Send} \mid \mathsf{Recv}) \times (\mathsf{Val} \to \mathsf{iProp})$ contain separation logic predicates over values, so we can make direct use of Iris's guarded recursion mechanism to define recursive protocols.

The reader may have noticed that we have already inserted the later modality $\triangleright$ in certain places in our definitions, such as in the definition of $c \overset{\mathsf{base}}{\rightarrowtail} p$ (Section 5.3.2). This is to make sure that $c \overset{\mathsf{base}}{\rightarrowtail} p$ is contractive in p, which in turn means that $!\,x\,\langle v \rangle \{P\}.\,p$ and $?\,x\,\langle v \rangle \{P\}.\,p$ are contractive in p. ⚙ We are therefore able to take guarded fixpoints of protocols, to create unbounded or infinite protocols, such as the following recursive variant of `prot_add`:

$$prot\_add\_rec \triangleq \mu p.\,!\,((\ell, x) : Addr \times \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\,?\,\langle () \rangle \{\ell \mapsto x + 2\}.\,p \qquad ⚙$$

A second component of guarded recursion is Iris's support for Löb induction. Löb induction allows us to verify unbounded or infinitely recursive programs that *use* recursive protocols. Ordinary induction only gives us an induction hypothesis for recursive calls where some measure is decreasing, and hence only works for terminating loops. Löb induction, on the other hand, gives us an induction hypothesis for *any* recursive call (not necessarily decreasing), but this induction hypothesis will be guarded under a later ($\triangleright$). These laters maintain logical consistency, but the resources guarded by them may only be accessed after the next primitive program step. In this manner, Löb induction allows us to verify partial correctness of a program that sends a stream of messages in an infinite tail-recursive loop, by instantiating the channel with the preceding recursive protocol.

The recursive protocols combined with Löb induction allow us to verify recursive programs such as the following recursive variant of the prog_add program from Section 5.2.3:

```
prog_add_rec ≜                                                    ⚙
  let (c₁, c₂) = new_imp () in
  fork {(rec f c₁ = let l = c₁.recv() in l ← (!l + 2); c₁.send(()); f c₁) c₁};
  let l = ref 38 in
  c₂.send(l); c₂.recv(); c₂.send(l); c₂.recv();
  assert(!l = 42); c₂
```

Here, **rec** $f x = e$ is a recursive function, where the recursive occurrence is bound to f. Verifying the program is straightforward. Notably, the main thread unfolds the recursive protocol *prot_add_rec* twice, to verify its code. The forked-off thread is resolved using Löb induction. It unfolds the recursive protocol once, verifies one iteration, after which it uses the Löb induction hypothesis to verify the recursive call. ⚙

Similar to Actris (Hinrichsen et al., 2022, §9.1), recursion is not only permitted via the tail p, but also via the proposition P in the protocols $? x \langle v \rangle \{P\}. p$ and $! x \langle v \rangle \{P\}. p$, making it possible to construct recursive protocols such as $\mu p. ! c \langle c \rangle \{c \rightarrowtail p\}. !\mathbf{end}$. We are allowed to construct such protocols because $c \rightarrowtail p$ is contractive in p. Also similar to Actris (Hinrichsen et al., 2022, §6.4), we can use Löb induction to prove that an infinitely recursive protocol is a subprotocol of another. The later modalities ($\triangleright$) in the rules for subprotocols (page 198) make it possible to remove a later from the Löb induction hypothesis. The same approach applies to protocols such as $\mu p. ! c \langle c \rangle \{c \rightarrowtail p\}. !\mathbf{end}$ because the subsumption rule $(c \rightarrowtail p) * \triangleright(p \sqsubseteq q) \twoheadrightarrow (c \rightarrowtail q)$ contains a later modality.

Making recursion and Löb induction interact properly requires careful placement of later modalities in the definitions of the channel points-to connectives. For example, to prove the subsumption rule $(c \rightarrowtail p) * \triangleright(p \sqsubseteq q) \twoheadrightarrow (c \rightarrowtail q)$ for other forms of channel closure in Section 5.5, we need to consider the case that $p = \mathbf{end}$ and

$q \neq \mathbf{end}$. We only obtain ▷ False from ▷$(p \sqsubseteq q)$, instead of an immediate contradiction (▷ False is not equivalent to False). Due to the later modalities in $c \rightarrowtail p$, however, ▷ False is sufficient to complete the proof.[3]

5.5 SELF-DUAL END

In the preceding sections, we had separate **close** and **wait** operations, with dual **!end** and **?end** protocols. In this section we investigate alternative operations to deallocate or close a channel, which result in a self-dual **end** protocol. We have two different options for achieving this:

- **Symmetric close.** Define one close operation, with protocol **end**, that both sides call, which dynamically determines who deallocates the channel (Section 5.5.1)

- **Send-close.** Define a combined send-close operation that sends the last message and closes the channel. The other side performs a **recv** that obtains no continuation channel (Section 5.5.2).

5.5.1 *Symmetric Close*

Suppose that we want only one **sym_close** operation, that both sides of the channel call. Because the channel consists of one memory location, we need to dynamically decide which caller gets to free the memory. We use compare-and-swap to achieve this effect:

$$\mathbf{sym\_close}\ c \triangleq \mathbf{if}\ \mathsf{CAS}(c, \mathbf{None}, \mathbf{Some}())\ \mathbf{then}\ ()\ \mathbf{else}\ \mathbf{free}\ c \qquad \text{⚙}$$

To see how this works, consider two parallel close operations on the same channel: **sym_close** c ∥ **sym_close** c. The thread that does its **CAS** first will successfully set c from **None** to **Some**(), and return () from its **sym_close**. The second thread will then *fail* its **CAS**, since the value stored in c is no longer **None**. It will then go to the else branch and free c.

To verify this version of close, we need to make a change to our notion of protocols. So far, our protocols have all been one-shot protocols $p \in \mathsf{Prot} \triangleq (\mathsf{Send}\ |\ \mathsf{Recv}) \times (\mathsf{Val} \rightarrow \mathsf{iProp})$ under the hood; even the protocols **!end**, **?end** $\in \mathsf{Prot}$. For the symmetric **sym_close**, this does not work. We now have to explicitly distinguish **end** in the protocols:

$$q \in \mathsf{Prot_{end}} ::= \mathbf{end}\ |\ p \qquad \text{where } p \in \mathsf{Prot} \qquad \text{⚙}$$

---

3 It also relies on ▷ False ⊢ $\boxed{R}^{\mathcal{N}}$ for any R and $\mathcal{N}$, see https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/897.

We also need to extend duality with $\overline{\textbf{end}} \triangleq \textbf{end}$ and subprotocols with $\textbf{end} \sqsubseteq \textbf{end}$. ✿ ✿ With this additional protocol, we have the following specification for `sym_close`:

Close operation: $\quad \{c \xrightarrow{\textbf{sym}} \textbf{end}\} \; \texttt{sym\_close} \; c \; \{\text{True}\}$ ✿

Because our set of protocols has been extended, we need an extended channel points-to $\xrightarrow{\textbf{sym}}$, which we define as follows:

$$c \xrightarrow{\textbf{sym}} q \triangleq \begin{cases} \exists \gamma_1, \gamma_2, \ell. \; \triangleright(c = \ell) * \boxed{\text{end\_inv } \gamma_1 \; \gamma_2 \; \ell} \; * \triangleright(\text{tok}\,\gamma_1) & \text{if } q = \textbf{end} \\ c \mapsto q & \text{if } q \in \text{Prot} \end{cases}$$

✿

Here, the following protocol is stored inside our invariant:

$$\text{end\_inv } \gamma_1 \; \gamma_2 \; \ell \triangleq \underbrace{(\ell \mapsto \textbf{None})}_{\text{before close}} \lor \underbrace{(\ell \mapsto \textbf{Some}() * (\text{tok}\,\gamma_1 \lor \text{tok}\,\gamma_2))}_{\text{one side has closed}} \lor \underbrace{(\text{tok}\,\gamma_1 * \text{tok}\,\gamma_2)}_{\text{fully closed}}$$

✿

Like the one-shot send-receive protocol, this protocol uses two tokens $\text{tok}\,\gamma_1$ and $\text{tok}\,\gamma_2$, which belong to the two $c \xrightarrow{\textbf{sym}} \textbf{end}$ assertions. Initially, the invariant states that the location ℓ points to **None**. When one side has successfully closed, the invariant states that ℓ points to **Some**(), and the invariant has collected the token of the side that has called close first (because this is nondeterministic, the invariant uses a disjunction $\text{tok}\,\gamma_1 \lor \text{tok}\,\gamma_2$). When both sides have closed, the invariant has both tokens, and no memory points-to (because the memory location has been deallocated). As before, we add later modalities (▷) in front of $c = \ell$ and $\text{tok}\,\gamma_1$ to support infinite protocols via guarded recursion (Section 5.4). With these definitions, we can prove the Hoare specification for the symmetric `sym_close` in a similar way we verified `send1` and `recv1`. ✿

5.5.2 *Send-Close*

From an operational point of view, the previous two methods for channel closing are a tiny bit disappointing, because for the last step, a memory location is allocated but not used to communicate any useful message. In this section we develop a channel closing mechanism where the close operation is integrated with the last message send.

This may sound strange at first sight, but upon investigating how channel closing typically works in examples, it hopefully starts to make more sense. Consider an example where party A is communicating a stream of messages to another party B, and A may at every point decide to end the stream. This can be accomplished by sending an additional Boolean along with each message, which determines whether

this is the last message or not. When it is the last message, the sender *does not* allocate a continuation channel, and sends () in place of the continuation channel. When the receiver receives a message, they have to inspect the Boolean to determine whether they got a continuation channel or not. This saves one memory allocation and synchronization compared to the previous methods. Similarly, in the example of Figure 30, we can eliminate the last interaction and synchronization by integrating the final acknowledgment with the closing of the channel.

While this saving is minor, we argue in favor of it for aesthetic reasons. If one wants to implement the one-shot API on top of the previous session channel API (*i.e.,* the other way around compared to what we have done so far), then a single shot communication would involve one real communication and then one extra allocation and communication to close the channel. We now present a channel closing mechanism with which one can implement one-shot channels on top of session channels with no additional synchronizations or allocations. Therefore, with this channel closing mechanism, session channels become a *purely logical* layer over one-shot channels. The implementation of this closing mechanism is very simple, namely the following **send_close** operation:

$$\textbf{send\_close}\ c\ v \triangleq \textbf{send1}\ c\ (v, ())\qquad\qquad \text{⚙}$$

There is no corresponding **wait** operation for the other side: as **send_close** simply does not allocate a continuation channel, the other side can use **recv**, which already deallocates the memory location. For the specification and verification of **send_close**, we use the same Prot_{end} protocols:

$$q \in \text{Prot}_{\text{end}} ::= \textbf{end}\ |\ p \qquad \text{where } p \in \text{Prot} \qquad \text{⚙}$$

We extend duality with $\overline{\textbf{end}} \triangleq \textbf{end}$ and subprotocols with $\textbf{end} \sqsubseteq \textbf{end}$. As before, we define a new channel points-to, this time for the send-close version:

$$c \xmapsto{\textbf{scl}} q \triangleq \begin{cases} \triangleright(c = ()) & \text{if } q = \textbf{end} \\ c \mapsto q & \text{if } q \in \text{Prot} \end{cases} \qquad \text{⚙}$$

For the **end** protocol, the channel points-to asserts that there is no channel, i.e., the channel is a unit value instead of a pointer to a memory location (this could also be implemented as a null pointer).

These are the specifications for the channel operations with **send_close**:

New: $\{\mathsf{True}\}\ \mathbf{new}\ ()\ \{c.\ c \rightarrowtail^{\mathbf{scl}} p * c \rightarrowtail^{\mathbf{scl}} \overline{p}\}$ ⚙

Send: $\{c \rightarrowtail^{\mathbf{scl}} (!\, x \langle v \rangle \{P\}.\, p) * P\ t\}\ \mathbf{send}\ c\ (v\ t)\ \{c'.\ c' \rightarrowtail^{\mathbf{scl}} p\ t\}$ where $p\ t \neq \mathbf{end}$
                                                                                   ⚙

Send-close: $\{c \rightarrowtail^{\mathbf{scl}} (!\, x \langle v \rangle \{P\}.\, p) * P\ t\}\ \mathbf{send\_close}\ c\ (v\ t)\ \{\mathsf{True}\}$ where $p\ t = \mathbf{end}$
                                                                                   ⚙

Receive: $\{c \rightarrowtail^{\mathbf{scl}} (?x \langle v \rangle \{P\}.\, p))\}\ \mathbf{recv}\ c\ \{w.\ \exists y, c'.\ w = (v\ y, c') * c' \rightarrowtail^{\mathbf{scl}} p\ y * P\ y\}$
                                                                                   ⚙

The **send** operation now requires that the tail $p\ t$ is *not* **end**, whereas the **send_close** operation requires that $p\ t$ *is* **end**. The specification of **recv** does not concern itself with **end**. Instead, the received message $v\ y$ will contain information about whether the protocol ended or not (such as a Boolean, as described previously). Using logical reasoning about the message, we can then conclude whether the tail protocol $p\ t$ is **end** or not. If it is, then we obtain $c' = ()$, and we do not need to do anything. If it is not **end**, we obtain $c' \rightarrowtail^{\mathbf{scl}} p\ y$ and can continue the protocol.

Unlike **close** with symmetric channel closing from Section 5.5.1, the **send_close** operation has been defined in terms of **send1**. The proofs of the specifications therefore also follow straightforwardly from the specifications of **send1** and **recv1**, unlike the proofs for symmetric channel closing.

5.6 OTHER SUPPORTED FEATURES

In this section, we briefly discuss some other features of our framework. Similar to Actris, we get these features for free by building on top of Iris:

DELEGATION AND CHANNEL PASSING   We support delegation, *i.e.,* sending channels over channels as messages, due to Iris's support for impredicative (*i.e.,* nested) invariants. This allows the channel points-to resource to be used in a protocol such as $!\, c \langle c \rangle \{c \rightarrowtail q\}.\, p$ This protocol enables us to send a channel c as well as its associated channel points-to $c \rightarrowtail q$ over another channel, which then allows the receiver to use the received channel c at protocol q.

CHOICE PROTOCOLS   We support choice protocols, where a thread can choose between multiple different continuation protocols. This can be encoded as a special case of dependent session protocols, where the sender makes the choice by sending a Boolean value, and the continuation protocol is chosen based on the value of the Boolean: $p_1 \oplus p_2 \triangleq !\, b \langle b \rangle \{\mathsf{True}\}.\ \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2$.

SHARED MEMORY   Channels are not the only way to communicate information between threads: we can also use shared memory directly. We can use all of the

features of Iris to reason about shared memory, we can send mutable references as messages over channels (as in Figure 30), and we can store channels in mutable references.

LOCKS AND SHARED SESSIONS   We support the combination of locks with channel communication. For instance, we can use a lock to protect a channel endpoint, which can then be used by multiple threads. This is useful for implementing shared sessions, where multiple threads can send and receive messages on the same channel endpoint, which is common in client-server protocols.

5.7 MECHANIZATION

The implementations of channels (Section 5.2), the proof that they satisfy their separation logic specifications (Section 5.3), the different methods for closing channels (Section 5.5), and the verification of all the examples have been fully mechanized using the Coq proof assistant (Coq Team, 2021), making use of the Iris separation logic framework.

The mechanization follows the layered design as presented in Figure 29. The layered design allows our proofs to be simpler compared to previous work on Actris (Hinrichsen et al., 2020). Only the proofs for one-shot operations `new1`, `send1`, `recv1` (and the symmetric `sym_close`) involve concurrent separation logic concepts such as ghost state and invariants. All the other proofs are done on top of these specifications, treating the one-shot operations as a black box.

Our protocol definitions are simple compared to Actris. We do not need to solve an intricate recursive domain equation (Hinrichsen et al., 2022, §9.7). At no point do we have to reason about more than one cell in the buffer structure; the multi-shot session protocols simply emerge automatically using composition. Despite this simplification to the Actris model, the different extensions such as subprotocols, guarded recursion, and the different forms of channel closing work seamlessly together. For instance, we can show that an infinitely recursive protocol is a subprotocol of another infinitely recursive protocol, by using guarded recursion and Löb induction.

In total, our Coq mechanization consists of less than 1000 lines of Coq code (including the verification of all examples). The mechanization is referenced throughout the paper by ✿-symbols. The mechanization has also been archived on Zenodo (Jacobs et al., 2023).

5.8 RELATED WORK

The origins of our line of work trace back to session types. More directly, our work is inspired by encodings of session types in terms of one-shot synchronization in particular (Kobayashi, 2002b; Dardha et al., 2017; Jacobs et al., 2022a). Our work is also directly related to dependent protocols and program logics for session protocols.

Most notable is the work on Actris (Hinrichsen et al., 2020, 2022), which introduced the notion of *dependent separation protocols*, which we use to specify our session channels. We go over each of these points in more detail below.

ONE-SHOT CHANNELS   The encoding of session channels in terms of sequenced one-shot channels originated in the π-calculus. This encoding sends a continuation channel in each message, so that the communication can continue. Kobayashi (2002b) showed that session types can be encoded into π-types, and Dardha et al. (2012, 2017) later extended Kobayashi (2002b)'s approach. Jacobs et al. (2022a) presented a bidirectional version in a λ-calculus.

Similar one-shot primitives have also been used in the implementation of message passing libraries, such as in the work of Scalas and Yoshida (2016a); Padovani (2017); Kokke and Dardha (2021b); Niehren et al. (2006). Our implementation of session channels in terms of one-shot channels uses a similar strategy.

Unlike this earlier work, which is either untyped or type-based, we use session protocols in separation logic to verify (partial) functional correctness. Our one-shot channels are not primitive and not built-in to the language, but implemented in terms of low-level memory operations. We take inspiration from the preceding work and subsequently build session channels on top of one-shot channels, and we build session protocols on top of one-shot protocols.

DEPENDENT PROTOCOLS AND SESSION LOGICS   Bocchi et al. (2010) and Toninho et al. (2011) both developed version of (multi-party) session types which incorporate logical binders into the protocols, alongside a first-order decidable assertion language for specifying properties about them. Later, Toninho and Yoshida (2018) and Thiemann and Vasconcelos (2020) expanded on this work by allowing similar binders determine the structure of the remaining protocol, similar to what we do in Section 5.2.4. Compared to our work, their assertion languages are limited in the sense that they cannot describe the delegation of resources (*e.g.*, sending a reference to another thread). Later work (Craciun et al., 2015; Costea et al., 2018) addressed the issue of specifying resource delegation, through the development of a *session logic*, based in separation logic. Their logic allows ascribing channel endpoints with protocols, which in turn can specify resources to be shared, such as other channel endpoints. Compared to our work, they do not support binders, which for one means that they cannot specify protocols referring the dynamically allocated references, like we do in Section 5.2.2. Actris protocols support both binders, delegation, and protocols referring to dynamically allocated references and ghost resources (Hinrichsen et al., 2020), as our protocols do.

ACTRIS   Actris introduced a shared-memory implementation of higher-order session channels, and the notion of dependent separation protocols for the verification of message passing concurrency using program logics, mechanized on top of Iris. Our work focuses primarily on developing a framework in the style of

Actris, but with a focus on *layered design*, *elegance*, and *simplicity*. This results in the following key differences between Actris and our work:

- Actris channels implement bi-directional communication using a pair of buffers that are protected by a lock. Our one-shot channels are implemented directly using load and store memory operations, and our session channels and imperative channels are implemented in terms of one-shot channels.

- As a result of this, Actris's dependent separation protocols are defined by solving an intricate recursive domain equation. By contrast, our definition of Prot ≜ (Send | Recv) × (Val → iProp) is itself non-recursive, yet Actris-style dependent separation protocols can be *defined* as inhabitants of Prot, and automatically support recursive protocols.

- Our notion of subprotocols for one-shot channels is very simple and non-recursive, but automatically lifts to (recursive) session protocols, because session protocols are defined as one-shot protocols. Actris's notion of subprotocols is recursive and more complicated than ours, but also stronger: Actris's implementation of channels with a pair of buffers admits swapping sends over receives (akin to asynchronous subtyping (Mostrous et al., 2009; Mostrous and Yoshida, 2015)). Such a transformation is not sound for our single-buffer implementation of channels.

- We achieve a simpler approach by making use of nested invariants, but Actris's solution gave rise to the "Actris ghost theory" (Hinrichsen et al., 2022, §9.4) for reasoning about session protocols in a way that is disconnected from specific implementations. The Actris ghost theory has been used to develop specifications based on dependent session protocols for distributed systems (Gondelman et al., 2023).

- Actris contains a number of convenience features, such as multi-binders and associated tactics, to ease verification of message passing programs in Coq. While such features can be integrated in our Coq development, we preferred to keep the protocols (and verification thereof) simpler, to focus on the layering of channel variants. Even so, our single-binders can simulate multi-binders using tuples, as has been demonstrated throughout the paper.

- While Actris relies on a garbage collector for channel deallocation, we present several manually memory managed solutions for channel closing.

In short, Actris has more features (asynchronous subtyping, ghost theory) and a more convenient implementation in Coq (multi-binders, tactics), but our design achieves the key feature of Actris (dependent separation protocols) in a conceptually simpler and layered manner: once we have defined and verified one-shot channels (which are quite simple and require only the simplest form of ghost resources to

verify), we treat them as a black box and develop Actris-style protocols with relative ease and without any further use of ghost state or invariants.

An application of Actris is the verification of the soundness of a session type system via the method of semantic typing (Hinrichsen et al., 2021). Since our separation logic specifications for session channels are the same as Actris's, a similar result could be achieved with our development.

IMPERATIVE SESSION CHANNELS   Related to Section 5.2.4, there has also been work on *type systems* for imperative channels, which free the user from having to thread channel variables through their program Saffrich and Thiemann (2023). The advantage of a type system compared to a program logic is that type checking is automatic, but an advantage of a program logic is its ability to verify functional correctness. Hinrichsen et al. (2021) combines advantages of both approaches via the method of semantic typing in Iris, which allows one to combine separation logic verification for intricate parts of the program, and type checking for the rest.

Chapter 6

Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing

ABSTRACT   We introduce a linear concurrent separation logic, called **LinearActris**, designed to guarantee deadlock and leak freedom for message-passing concurrency. LinearActris combines the strengths of session types and concurrent separation logic, allowing for the verification of challenging higher-order program with mutable state through dependent protocols. The key challenge is to prove the adequacy theorem of LinearActris, which says that the logic indeed gives deadlock and leak freedom "for free" from linearity. We prove this theorem by defining a step-indexed model of separation logic, based on *connectivity graphs*. To demonstrate the expressive power of LinearActris, we prove soundness of a higher-order (GV-style) session type system using the technique of logical relations. All our results and examples have been mechanized in Coq.

6.1 INTRODUCTION

Session type systems (Honda, 1993; Honda et al., 1998) allow type checking programs that involve message-passing concurrency. Session types are protocols, which can be seen as sequences of send (**!**) and receive (**?**) actions. They are associated with channels, and express in what order messages of what type should be transferred. For example, the session type **!Z.?B.end** is given to a channel over which an integer should be sent, after which a boolean is received. More complex session types can be formed with operators for choice (⊕,&), recursion (μ), *etc.*

   Aside from ensuring type safety, linear session type systems (Caires and Pfenning, 2010; Wadler, 2012) can ensure deadlock freedom. That means that well-typed programs cannot end up in a state where all threads are waiting to receive a message from another. Deadlock freedom has been extended to large variety of session type systems (Carbone and Debois, 2010; Fowler et al., 2021; Toninho et al., 2013; Toninho, 2015; Caires et al., 2013; Pérez et al., 2014; Lindley and Morris, 2015, 2016a, 2017; Fowler et al., 2019; Das et al., 2018). The elegance of session type systems is that they give deadlock freedom essentially "for free"—it is obtained from "just" linear type checking. Moreover, session types are compositional—once functions have been type checked, they can be composed by merely establishing that the types agree. A final strength of session types is that deadlock freedom is maintained in a higher-order setting where closures and channels are transferred as first-class data over channels. *The goal of this chapter is to extend these advantages to separation logic.*

The key aspect that makes session types unique and different from other methods for deadlock freedom—such as lock orders (Dijkstra, 1971; Leino et al., 2010; Hamin and Jacobs, 2018; Balzer et al., 2019; D'Osualdo et al., 2021b), priorities (Kobayashi, 1997; Padovani, 2014; Dardha and Gay, 2018), and global multiparty session types (Honda et al., 2008, 2016)—is that linear session types do not require any additional proof obligations involving orders, priority annotations, or global types. Still, other methods neither supersede nor subsume session types in the range of programs they can prove to be deadlock free. This will be further discussed in Section 6.8.1.

The ideas of session types are not limited to type checking, but have previously also been applied to functional verification. Bocchi et al. (2010); Craciun et al. (2015); Hinrichsen et al. (2020, 2022) have developed program logics that incorporate concepts from session types to verify increasingly sophisticated programs with message-passing concurrency. The protocols of these program logics make it possible to put logical conditions on the messages, allowing one to specify the contents (*e.g.,* the message is an even number) instead of just the shape (*e.g.,* it is an integer). The state of the art is the Actris logic and its descendants (Hinrichsen et al., 2020, 2022; Jacobs et al., 2023), which are embedded in the Iris framework for concurrent separation logic in Coq (Jung et al., 2015, 2016; Krebbers et al., 2017a; Jung et al., 2018b). Actris' dependent separation protocols can express dependencies between the data of messages and specify the transfer of resources. For example, the protocol $!(\ell : \mathsf{Addr}, n : \mathbb{N})\langle\ell\rangle\{\ell \mapsto n\}; \mathbf{?}\langle n\rangle\{\ell \mapsto (n + 1)\}; \mathbf{end}$ says that a location ℓ with value n should be sent, after which the value n should be received, and the value of ℓ has been incremented.

Since Actris is a full-blown program logic, instead of a type system that aims to have decidable type checking, it can express more protocols and therefore verify safety of more programs than session types. In particular, it can express protocols where the shape (*e.g.,* number of messages) of the protocol depends on the contents of earlier messages. Moreover, Hinrichsen et al. (2021) show that Actris can be used to give a semantic model to prove soundness of (affine) session types using the technique of logical relations in Iris (Timany et al., 2022).

A key ingredient of concurrent separation logics such as Iris (on top of which Actris is built)—and also other separation logic frameworks such as VST (Appel, 2014), CFML (Charguéraud, 2020), and BedRock (Chlipala, 2013)—is their *adequacy* (a.k.a. soundness) theorem that connects the program logic to the operational semantics. For Iris, the adequacy theorem is (Jung et al., 2018b, §6.4):

> A closed proof of {True} e {True} implies that e is **safe**, *i.e.,* if $([e], \emptyset) \leadsto^*$ $([e_1 \ldots e_n], h)$, then for each i either e_i is a value or (e_i, h) can step.

Intuitively this theorem says that the logic is doing its job: a verified program e "cannot go wrong", *i.e.,* it cannot perform illegal operations such as loading from a dangling location (use after free) or use an operator with wrong arguments (*e.g.,* $3 + \lambda x.x$). Formally it says that if e can be verified (*i.e.,* a Hoare triple with trivial precondition can be proved), and the initial configuration $([e], \emptyset)$ (consisting of a

single thread e and the empty heap) steps to $([e_1 \ldots e_n], h)$ (consisting of threads $e_1 \ldots e_n$ and heap h), then each thread e_i has either finished (is a value) or can make further progress (can perform a step). Illegal operations cannot step, so adequacy guarantees they do not occur.

Despite the strong trust that the adequacy theorem gives in the correctness of a program logic—especially when mechanized in a proof assistant such as Coq—the adequacy theorem of most state-of-the-art program logics says nothing about deadlocks. In Iris, blocking operations (*e.g.*, receiving from a channel whose buffer is empty, or acquiring a lock that has already been acquired) are modeled as busy loops, and thus can always step, and are trivially safe.

GOAL OF THE PAPER. The goal of this chapter is to build a program logic that (1) enjoys an adequacy theorem that guarantees deadlock freedom for message passing concurrency, (2) combines the strengths of session types and concurrent separation logic to obtain deadlock freedom "for free" from linearity, without any additional proof obligations, and (3) is strong enough to verify challenging programs. Before discussing the desiderata of the program logic, let us investigate the operational semantics and adequacy theorem. To distinguish between deadlock and non-termination, receiving from a channel blocks the thread until a message is sent, instead of performing a busy loop. With that change at hand, the adequacy theorem becomes similar to the global progress theorem of session type systems (Caires and Pfenning, 2010):

> A closed proof of $\{\mathsf{Emp}\}\ e\ \{\mathsf{Emp}\}$ implies that e enjoys **global progress**, *i.e.*, if $([e], \emptyset) \rightsquigarrow^* ([e_1 \ldots e_n], h)$, then either e_i is a value for each i and $h = \emptyset$, or $([e_1 \ldots e_n], h)$ can step.

Instead of requiring each thread to step, which would be false if a thread is genuinely waiting for another thread, we require the configuration as a whole to step. This means that there is always at least one thread that can step, *i.e.*, there is no global deadlock. Additionally, compared to the adequacy theorem for safety, we require the final heap to be empty, which means all channels have been deallocated, *i.e.*, there are no memory leaks. (Note that global progress does not subsume safety, we still need a theorem that ensures the absence of illegal non-blocking operations.)

Our desired adequacy theorem does *not* hold for Iris-based logics such as Actris:

- **The need for linearity.** Iris and Actris are *affine*, which means that resources must be used at most once, but can also be dropped (Iris satisfies the proof rule $P * Q \vdash P$, or equivalently $\mathsf{Emp} \dashv\vdash \mathsf{True}$). Hence one can verify a program that creates a channel with endpoints c_1 and c_2, have one thread perform a receive, and let the other thread perform a no-op:

$$\textbf{Thread 1:} \quad c_1.\textbf{recv}()$$
$$\textbf{Thread 2:} \quad \text{do nothing}$$

This program can be verified in Iris/Actris because using affinity, the ownership of c_2 can be dropped in the second thread. However, this program causes a deadlock: due to the absence of a send, the receive will block indefinitely. In session types this form of deadlock is ruled out by making the system *linear*, which means that resources must be used exactly once, and cannot be dropped until the protocol has been completed.

- **The need for acyclicity** Linearity alone is not enough. If a thread could obtain ownership of both endpoints of a single channel, then it would be able to trivially deadlock itself, by performing the receive before the send. Linearity would not be violated, as the thread would still consume both channel ownership assertions according to the rules of the logic, but the thread would be blocked forever. More generally, if two threads own the endpoints of two channels, and perform a receive followed by send, there would be a deadlock:

$$\textbf{Thread 1:} \quad c_1.\textbf{recv}(); \; d_1.\textbf{send}(2)$$

$$\textbf{Thread 2:} \quad d_2.\textbf{recv}(); \; c_2.\textbf{send}(1)$$

In session types, Wadler (2012) addresses this problem by combining thread and channel creation into a single construct. Together with linearity, this ensures that channel ownership is *acyclic* in a certain sense, and rules out all deadlocks without need for annotations.

In this chapter we introduce **LinearActris**—which amends Actris with the aforementioned restrictions from linear session types outlined to satisfy the goals we stated above. The key challenge that we address in the remainder of the introduction is proving the adequacy theorem of LinearActris.

KEY CHALLENGE: PROVING ADEQUACY   Adequacy is commonly proved by giving a semantic interpretation of propositions and Hoare triples. For sequential separation logic (O'Hearn et al., 2001), propositions are modeled as heap predicates, and the semantics of Hoare triples is defined so that safety and leak freedom follow almost by definition. Since we consider a higher-order program logic, for a concurrent language with dynamic thread and channel spawning, and wish to prove global progress, this simple setup no longer suffices. We list the challenges below:

- **Circular semantics.** Session types and dependent separation protocols of Actris are higher-order, which means they can specify programs that transfer channels and closures over channels. In Actris one can write $d \rightarrowtail \; !(c : Addr)\langle c \rangle \{c \rightarrowtail p\}; \textbf{end}$ to say that d is a channel, over which a channel c with protocol p is sent. Here, the protocol p can contain protocol ownership assertions $c \rightarrowtail p'$, where p' can in turn contain protocol ownership assertions. This circularity involves a negative recursive occurrence and cannot be solved in set theory. It is similar to the type-world circularity in models of type systems with higher-order references (Ahmed, 2004; Birkedal et al., 2011), and that of

storable locks (Hobor et al., 2008) and impredicative invariants (Svendsen and Birkedal, 2014), where step-indexing (Appel and McAllester, 2001) is used to solve the circularity. The original Actris makes (in part) use of Iris's impredicative invariant mechanism to avoid solving this circularity explicitly.

- **Invariants and linearity.** Unfortunately, Iris's invariants are strongly tied to the logic being affine. Jung (2020, Thm 2) presents a paradox showing that a naïve linear version of Iris's invariants cannot be used to obtain even leak-freedom. Bizjak et al. (2019) present Iron, a linear version of Iris with an invariant mechanism that can be used in to prove leak-freedom. Aside from not considering deadlock freedom, Iron avoids Jung's paradox by restricting the contents of invariants—namely, invariants cannot contain permissions to deallocate resources. Ownership of the **end** protocol needs to provide permission to deallocate the channel, making Iron's invariants insufficient for our purpose.

- **Invariants and acyclicity.** Linearity alone is not enough to avoid deadlocks—one needs to maintain an invariant that the channel ownership topology is acyclic. Formalizing this acyclicity invariant is a key challenge of the syntactic meta theory of session types (Lindley and Morris, 2015, 2016a; Fowler et al., 2021; Jacobs et al., 2022a,b). Since this prior work is aimed at syntactic theory of type systems, we need to investigate how to incorporate acyclicity of the topology into a semantic model of a program logic. Additionally, in type systems there is a 1-to-1 correspondence between physical references and ownership, but not in program logics. One can create protocols such as **!**$(c : Addr)\langle c\rangle\{c \rightarrowtail p\}$; **?**$\langle()\rangle\{c \rightarrowtail p'\}$; **end** where a channel reference and ownership is sent, and only an acknowledgment () is returned. This means that the sending thread has to keep a reference to the channel, although it cannot use it before it has received the acknowledgment.

We define a step-indexed linear model of separation logic as the solution of a recursive domain equation (America and Rutten, 1989; Birkedal et al., 2010). To avoid reasoning about step-indices, we work in the pure step-indexed logic with a later modality (▷) (Appel et al., 2007; Nakano, 2000).

Similar to Iris we define Hoare triples in terms of weakest preconditions (Krebbers et al., 2017a). A major difference in the definition of the weakest precondition compared to Iris is that we thread through the weakest preconditions of all threads, as well as the ownership and duality invariants of all channels. This way we can ensure that at all times the threads and channels form an acyclic topology with respect to channel ownership. To formalize acyclicity we use the notion of a *connectivity graph* by Jacobs et al. (2022a). To simplify the construction of the model and the operational semantics of the language, we base ourselves on the work of Dardha et al. (2012); Jacobs et al. (2023): we use one-shot channels as primitive, and build multi-shot channels on top of those.

CONTRIBUTIONS   We introduce **LinearActris**—a concurrent separation logic for proving deadlock- and leak freedom of message-passing programs, essentially offering these guarantees "for free" from linearity, without any additional proof obligations. This involves the following contributions:

- We verify a range of examples of that use channels, closures, and mutable references as first-class data, demonstrating the expressive power of LinearActris (Section 6.2).

- We provide a formal description of the proof rules of LinearActris. First for multi-shot channels, and then for one-shot channels. Based on Jacobs et al. (2023), we derive the logic for multi-shot channels from the one for one-shot channels (Sections 6.3 and 6.4).

- We provide a formal adequacy proof of LinearActris based on a step-indexed model of separation logic rooted in connectivity graphs (Jacobs et al., 2022a), showing that a derivation in LinearActris ensures deadlock and leak freedom of the program in question (Sections 6.5 and 6.6).

- To demonstrate an application that truly relies on our connectivity based approach to deadlock freedom (and is out of scope for logics based on *e.g.,* lock orders), we construct a logical relations models in LinearActris that establishes deadlock freedom for a session-typed language that goes beyond GV-like systems, and supports recursive types, subtyping, term- and session type polymorphism, and unique mutable references (Section 6.7).

- We have mechanized all our results in Coq. We provide custom tactics for reasoning in LinearActris, built on top of the Iris Proof Mode (Krebbers et al., 2017b, 2018). We have used these tactics to prove deadlock and leak freedom of all of our examples (Section 6.2) and those of §1,5,6.3,10 of Actris 2.0 (Hinrichsen et al., 2022), as well as in the proofs of the logical relation (Section 6.7). See the Coq mechanization for the sources, comprising 13.341 lines of Coq code.

6.2   LINEAR ACTRIS BY EXAMPLE

In this section we present LinearActris with example programs that we want to verify. We deliberately use very small examples. In our Coq mechanization we show that LinearActris can also be used to prove deadlock freedom of more challenging examples from the Actris papers, in particular, a number of increasingly complicated versions of parallel merge sort. ⚙

The programming language that we use in LinearActris is called ChanLang. It has concurrency, bidirectional message passing channels, mutable references, and functional programming constructs (such as lambdas, products, sums, and recursion). The syntax is shown in Figure 37. ChanLang has the following constructs

$$e \in \text{Expr} ::= x \mid e\,e \mid \lambda x.e \mid (e,e) \mid \text{inl } e \mid \text{inr } e \mid \textbf{rec } f\,x = e \mid n \mid e + e \mid \cdots$$

<div style="text-align:right">✿</div>

$$\mid \; \textbf{match } e \textbf{ with } \text{inl } x \Rightarrow e; \text{ inr } x \Rightarrow e \textbf{ end} \mid \textbf{let } (x_1, x_2) = e \textbf{ in } e$$

$$\mid \; \textbf{fork1 } e \mid e.\textbf{send}(e) \mid e.\textbf{recv}() \mid e.\textbf{close}() \mid e.\textbf{wait}()$$

(Channel operations)

$$\mid \; \textbf{ref } e \mid \,!\,e \mid e \leftarrow e \mid \textbf{free } e \mid \textbf{assert}(e) \qquad \text{(Heap operations \& assert)}$$

Figure 37: The syntax of ChanLang.

for message-passing concurrency:

fork $(\lambda c.\ e)$ Fork a new thread for e with channel c, return the channel.

   $c.\textbf{send}(v)$ Send message v over the channel c.

   $c.\textbf{recv}()$ Receive a message over the channel c.

   $c.\textbf{close}()$ Close the channel c.

   $c.\textbf{wait}()$ Wait for the channel c to be closed.

The $c.\textbf{recv}()$ and $c.\textbf{wait}()$ operations are blocking, and could thus potentially lead to deadlocks. As is common in session-typed languages like GV (Wadler, 2012; Gay and Vasconcelos, 2010), our fork operation both spawns the child thread, and sets up a channel for communication between the parent thread and child thread. This will turn out to be important for deadlock freedom (Sections 6.5 and 6.6).

The following example illustrates how we can use these constructs to fork off a thread that receives a message from the main thread, adds one to it, and sends it back:

let $c_1 =$ **fork** $(\lambda c_2.\ c_2.\textbf{send}(c_2.\textbf{recv}() + 1);\ c_2.\textbf{close}())$ **in**   ✿

$c_1.\textbf{send}(1);\ \textbf{assert}(c_1.\textbf{recv}() == 2);\ c_1.\textbf{wait}()$

The $\textbf{assert}(e)$ operation asserts that e evaluates to true, and otherwise it gets stuck. Illegal operations more generally, such as sending over a closed channel, also get stuck forever. To verify the program, we need to reason about the channels c_1 and c_2. We do so by means of channel ownership assertions $c \rightarrowtail p$, which state that we own a reference to the channel c, and we must interact with it according to the protocol p. Our protocols are *dependent separation protocols* in the style of Actris (Hinrichsen et al., 2020). We can use the following dual pair of protocols for c_1 and c_2 at the **fork**:

$$c_1 \rightarrowtail \,!\langle 1 \rangle;\ ?\langle 2 \rangle;\ ?\textbf{end} \qquad\qquad c_2 \rightarrowtail \,?\langle 1 \rangle;\ !\langle 2 \rangle;\ !\textbf{end} \qquad ✿$$

In these protocols, each step is either $!\langle v \rangle$ or $?\langle v \rangle$, indicating that the owner of the reference must **send** or **recv** a value v, respectively. The final $!\textbf{end}$ / $?\textbf{end}$ indicates that the protocol is finished, and that the **close** / **wait** operation must be performed.

QUANTIFIED PROTOCOLS   The preceding protocol is inflexible, because it specifies the exact values that must be sent and received. To alleviate this inflexibility, we can use a *quantified protocol* instead:

$$c_1 \rightarrowtail \ !(n : \mathbb{N})\langle n \rangle; \ ?\langle n + 1 \rangle; \ ?\textbf{end} \qquad\qquad c_2 \rightarrowtail \ ?(n : \mathbb{N})\langle n \rangle; \ !\langle n + 1 \rangle; \ !\textbf{end} \quad \text{⚙}$$

This protocol states that if we send n, then we will receive $n + 1$. When verifying a quantified protocol step, the sender can instantiate the quantified variable with any logical value. For example, the sender can instantiate n with 1, and send 1 over the channel. The receiver must be verified to work for any n chosen by the sender. The continuation of the protocol is allowed to be an arbitrary function of the quantified variables. This can be used to verify examples such as the following:

```
let n = c.recv() in if n < 5 then c.close() else c.send(n − 5); . . .    ⚙
```

The protocol for c will have to have a different length, depending on which branch of the **if** is taken. We can verify this program using the following protocol for c:

$$c \rightarrowtail \ ?(n : \mathbb{N})\langle n \rangle; \ \textbf{if} \ n < 5 \ \textbf{then} \ (!\textbf{end}) \ \textbf{else} \ (!\langle n - 5 \rangle; \ \ldots) \qquad \text{⚙}$$

MUTABLE REFERENCES   In addition to channels, our language has mutable references:

ref v	Allocate a new location in the heap and store the value v in it.
$!\ell$	Read the value from the location ℓ.
$\ell \leftarrow v$	Write the value v to the location ℓ.
free ℓ	Free the location ℓ.

Illegal operations, such as using a location that has been freed, are modeled as getting stuck forever. Consider the following variant of the preceding example:

```
let c₁ = fork (λc₂. let l = c₂.recv() in l ← !l + 1; c₂.close()) in    ⚙
let l = ref 1 in c₁.send(l); c₁.wait(); assert(!l == 2); free l
```

We send a mutable reference from the main thread to the forked thread, which increments it. The main thread waits for the forked thread to close its channel, and then asserts that the value of the reference is 2. The reference is then freed by the main thread. LinearActris can prove that this program is safe and does not deadlock. Note that the safety relies on the blocking behavior of $c_1.\textbf{wait}()$, which ensures that

the forked thread has finished before the main thread asserts that the value is 2 and frees the reference. The protocols to verify this program are as follows:

$$c_1 \rightarrowtail \ !(l : \text{Addr}, n : \mathbb{N})\langle l \rangle \{l \mapsto n\}; \ ?\textbf{end}\{l \mapsto n + 1\}$$
$$c_2 \rightarrowtail \ ?(l : \text{Addr}, n : \mathbb{N})\langle l \rangle \{l \mapsto n\}; \ !\textbf{end}\{l \mapsto n + 1\}$$

This time, the protocol is parameterized by both the location l and the value n that is initially stored in the location. The protocol states that if we send a location l, then this location will be incremented by 1. The curly brackets $\{\_\}$ indicate the separation logic resources that are sent along with the message. In the protocol for c_1 above, the heap ownership assertion $l \mapsto n$ is transmitted with the initial **send** step, and $l \mapsto n + 1$ is received in the **wait** step. As the following example shows, a reference need not be send over the channel, but can also be captured by the closure:

```
let l = ref 1 in let c₁ = fork (λc₂. l ← !l + 1; c₂.close()) in
c₁.wait(); assert(!l == 2); free l
```

We transfer $l \mapsto 1$ to the child thread immediately upon the **fork**, and the protocols simplify to:

$$c_1 \rightarrowtail \ ?\textbf{end}\{l \mapsto 2\} \qquad c_2 \rightarrowtail \ !\textbf{end}\{l \mapsto 2\}$$

SENDING CHANNELS OVER CHANNELS   In addition to exchanging references, LinearActris can also reason about programs that send channels over channels. Consider the following example:

```
let d₁ = fork (λd₂. assert(d₂.recv().recv() == 2); d₂.close()) in
let c₁ = fork (λc₂. c₂.send(2); c₂.wait()) in
d₁.send(c₁); d₁.wait(); c₁.close()
```

The program forks off two threads, which gives the main thread two channels c_1 and d_1. The main thread then sends c_1 over d_1, and waits for d_1 to be closed, and then closes c_1. The first thread receives c_1 from d_2, and then receives on c_1 and asserts that the value is 2, and then closes d_2. The second thread sends 2 over c_2, and then waits for c_2 to be closed.

That this program is safe and does not deadlock can be proven by LinearActris, but this is more subtle than one might think: if we were to swap the two d_1.**wait**(); c_1.**close**() operations, then the program would not be safe, as c_1 might be

closed before the other threads are done with it. We can verify the example using the following protocols:

$$c_1 \rightarrowtail \text{?}\langle 2 \rangle; \text{ !end} \qquad c_2 \rightarrowtail \text{!}\langle 2 \rangle; \text{ ?end} \qquad \text{⚙}$$

$$d_1 \rightarrowtail \text{!}(c : \text{Addr})\langle c \rangle\{c \rightarrowtail \text{?}\langle 2 \rangle; \text{ ?end}\}; \text{ ?end}\{c \rightarrowtail \text{!end}\} \qquad \text{⚙}$$

$$d_2 \rightarrowtail \text{?}(c : \text{Addr})\langle c \rangle\{c \rightarrowtail \text{?}\langle 2 \rangle; \text{ ?end}\}; \text{ !end}\{c \rightarrowtail \text{!end}\}$$

The protocol for c_1 and c_2 is simple: we send 2 and then end. The protocol for d_1 is more interesting: we send a (quantified) location c, and also send channel ownership for c, with the same protocol as we chose for c_1. The continuation of the protocol is **?end**$\{c \rightarrowtail \text{!end}\}$, which transfers ownership of c back to the main thread, but now at a new protocol.

STORING CHANNELS IN REFERENCES   Consider the following variation of the previous example, in which we wrap channel c_1 in a reference:

```
let d₁ = fork (λd₂. assert((! d₂.recv()).recv() == 2); d₂.close()) in          ⚙
let l = ref fork (λc₂. c₂.send(2); c₂.wait()) in
d₁.send(l); d₁.wait(); (! l).close(); free l
```

We can verify this example using the following protocol:

$$d_1 \rightarrowtail \text{!}(l : \text{Addr}, c : \text{Addr})\langle l \rangle\{l \mapsto c * c \rightarrowtail \text{?}\langle 2 \rangle; \text{ !end}\}; \text{ ?end}\{l \mapsto c * c \rightarrowtail \text{!end}\}$$

⚙

Wrapping the channel in a reference would allow the child thread to replace the contents of the reference with another channel entirely, as long as it satisfies the right protocol. For instance, the child thread could replace the channel in l with a new channel that it just created:

```
(! l).close(); l ← fork (λc. c.wait())
```

The c.**wait**() of the newly created channel can then match up with the (! l).**close**() of the main thread. To prevent leaks, the child thread must also close the channel originally in l. This variation can be verified with the same protocols as before.

SENDING CLOSURES   LinearActris can reason about higher-order programs that send closures that capture references and channels. Consider the following program, which spawns a thread that receives and runs a closure from the main thread, and then sends the result back:

```
let c₁ = fork (λc₂. let f = c₂.recv() in c₂.send(f ()); c₂.close()) in ···          ⚙
```

The protocol for c_1 is as follows:

$$c_1 \rightarrowtail \;!(f : Val, \Phi : Val \to aProp)\langle f \rangle \{WP\ f\ ()\ \{\Phi\}\};\; ?(v : Val)\langle v \rangle \{\Phi\ v\};\; \textbf{?end} \quad \text{⚙}$$

The protocol allows us to send a closure f, provided we also send a weakest precondition assertion WP f () {Φ}, which ensures that the return value v of f satisfies $\Phi\ v$. We can then receive v, and obtain the resources $\Phi\ v$. This protocol allows the closure f to capture linear resources in its environment, such as channels and references.

6.3 THE PROOF RULES OF LINEAR ACTRIS

In this section we present the rules of LinearActris. Similar to Iris, LinearActris is based on the weakest precondition WP e {Φ} connective, which is a separation logic assertion that intuitively states that if the program e is executed in the current heap, then its return value will satisfy predicate Φ. The Hoare triple {P} e {Φ} is syntactic sugar for P ⊢ WP e {Φ}. The adequacy theorem of LinearActris (Theorem 6.5.4) guarantees safety, deadlock freedom, and leak freedom for e provided we have a closed proof of Emp ⊢ WP e {Emp}.

6.3.1 *Basic Separation Logic*

Figure 38 displays the grammar of LinearActris propositions, as well as the basic rules for reasoning about weakest preconditions that involve pure expressions and mutable references. The WP rules in this figure are fairly standard, so we will only give a brief overview of them here. The rules WP-PURE-STEP and WP-VAL are the basic rules for reasoning about pure expressions. The rules WP-LÖB and WP-REC are used to reason about recursive functions. The WP-BIND rule is used to reason about an expression nested inside a (call-by-value) evaluation context. The WP-WAND rule can be used to weaken the postcondition, as well as to frame away parts of the precondition. The rules WP-ALLOC, WP-LOAD, WP-STORE, and WP-FREE reason about mutable references. In combination with inference rules for the logical connectives (which are not shown in the figure), these rules handle single-threaded programs, such as programs that manipulate mutable linked lists.

LINEARITY   An important distinction between LinearActris and logics like Iris, is that LinearActris is *linear* whereas Iris is *affine*. This means that in LinearActris, the rule P ⊢ Emp does not hold for all P, whereas in Iris it does.[1] This distinction is important, because this rule can be used to leak resources, for instance if P = $\ell \mapsto v$ then $\ell \mapsto v$ ⊢ Emp can be used to leak the location ℓ. LinearActris, unlike Iris, guarantees leak freedom, and thus forces us to **free** locations. Furthermore, as we

---

1 Logically equivalent formulations of the affine rule are P * Q ⊢ P or Emp ⊣⊢ True.

Separation logic propositions: ⚙

$$P, Q \in aProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \qquad \text{(Propositional logic)}$$
$$\mid \forall x.\, P \mid \exists x.\, P \mid x = y \qquad \text{(Higher-order logic with equality)}$$
$$\mid P * Q \mid P \twoheadrightarrow Q \mid \text{Emp} \qquad \text{(Separation logic)}$$
$$\mid \triangleright P \mid \text{WP } e\, \{\varPhi\} \qquad \text{(Step indexing and weakest preconditions)}$$
$$\mid \ell \mapsto v \mid \ell \rightarrowtail p \qquad \text{(Heap cell and channel ownership)}$$

Basic WP rules: ⚙

$$
\text{WP-pure-step} \quad
\frac{e_1 \rightsquigarrow_{\text{pure}} e_2 \qquad \text{WP } e_2\, \{\varPhi\}}{\text{WP } e_1\, \{\varPhi\}} *
\qquad
\text{WP-val} \quad
\frac{\varPhi\, v}{\text{WP } v\, \{\varPhi\}} *
\qquad
\text{WP-wand} \quad
\frac{\text{WP } e\, \{\varPhi\} \quad * \quad \forall v.\, \varPhi\, v \twoheadrightarrow \varPsi\, v}{\text{WP } e\, \{\varPsi\}} *
$$

$$
\text{WP-rec} \quad
\frac{\text{WP } e[v/x][\mathbf{rec}\ f\, x = e/f]\, \{\varPhi\}}{\triangleright \text{WP } (\mathbf{rec}\ f\, x = e)\, v\, \{\varPhi\}} *
\qquad
\text{WP-Löb} \quad
\frac{\triangleright P \twoheadrightarrow P}{P} \square
\qquad
\text{WP-bind} \quad
\frac{\text{WP } e\, \{v.\, \text{WP } K[v]\, \{\varPhi\}\}}{\text{WP } K[e]\, \{\varPhi\}} *
$$

Heap manipulation rules: ⚙

$$
\text{WP-alloc} \quad
\frac{}{\text{WP } \mathbf{ref}\, v\, \{\ell.\, \ell \mapsto v\}} *
\qquad
\text{WP-load} \quad
\frac{\ell \mapsto v}{\text{WP } !\ell\, \{v.\, \ell \mapsto v\}} *
\qquad
\text{WP-store} \quad
\frac{\ell \mapsto v}{\text{WP } \ell \leftarrow w\, \{\ell \mapsto w\}} *
$$

$$
\text{WP-free} \quad
\frac{\ell \mapsto v}{\text{WP } \mathbf{free}\, \ell\, \{\text{Emp}\}} *
$$

Figure 38: The basic rules of our separation logic.

Dependent separation protocols: ⚙

$$p \in \mathsf{Prot} ::= \ !(\vec{x})\langle v\rangle\{P\};\ p \qquad\qquad\qquad \text{(Send protocol)}$$
$$|\ ?(\vec{x})\langle v\rangle\{P\};\ p \qquad\qquad\qquad \text{(Receive protocol)}$$
$$|\ !\mathbf{end}\{P\}\ |\ ?\mathbf{end}\{P\} \qquad\qquad \text{(Close and wait protocol)}$$
$$|\ \mu\alpha.\ p\ |\ \alpha \qquad\qquad\qquad\qquad \text{(Recursive protocol)}$$

Duality and subprotocols: ⚙

$$\overline{!(\vec{x})\langle v\rangle\{P\};\ p} = ?(\vec{x})\langle v\rangle\{P\};\ \overline{p} \qquad \overline{!\mathbf{end}\{P\}} = ?\mathbf{end}\{P\}$$
$$\text{(Dual on dependent protocols)}$$
$$\overline{?(\vec{x})\langle v\rangle\{P\};\ p} = !(\vec{x})\langle v\rangle\{P\};\ \overline{p} \qquad \overline{?\mathbf{end}\{P\}} = !\mathbf{end}\{P\}$$

Sub-recv
$$\frac{\forall x_1.\ P_1\ x_1 \twoheadrightarrow \exists x_2.\ (v_1\ x_1 = v_2\ x_2) * P_2\ x_2 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{?(x_1)\langle v_1\rangle\{P_1\};\ p_1 \sqsubseteq ?(x_2)\langle v_2\rangle\{P_2\};\ p_2} *$$

Sub-wait
$$\frac{P_1 \twoheadrightarrow P_2}{?\mathbf{end}\{P_1\} \sqsubseteq ?\mathbf{end}\{P_2\}} *$$

Sub-send
$$\frac{\forall x_2.\ P_2\ x_2 \twoheadrightarrow \exists x_1.\ (v_2\ x_2 = v_1\ x_1) * P_1\ x_1 * \triangleright(p_1\ x_1 \sqsubseteq p_2\ x_2)}{!(x_1)\langle v_1\rangle\{P_1\};\ p_1 \sqsubseteq !(x_2)\langle v_2\rangle\{P_2\};\ p_2} *$$

Sub-close
$$\frac{P_2 \twoheadrightarrow P_1}{!\mathbf{end}\{P_1\} \sqsubseteq !\mathbf{end}\{P_2\}} *$$

Sub-chan
$$\frac{\triangleright p_1 \sqsubseteq p_2 \quad * \quad c \rightarrowtail p_1}{c \rightarrowtail p_2} *$$

Channel weakest precondition rules: ⚙

WP-fork
$$\frac{\forall c.\ (c \rightarrowtail \overline{p}) \twoheadrightarrow \mathsf{WP}\ e\ c\ \{\mathsf{Emp}\}}{\mathsf{WP}\ \mathbf{fork}\ e\ \{c.\ c \rightarrowtail p\}} *$$

WP-send
$$\frac{P[\vec{t}/\vec{x}] \quad * \quad c \rightarrowtail !(\vec{x})\langle v\rangle\{P\};\ p}{\mathsf{WP}\ c.\mathbf{send}(v[\vec{t}/\vec{x}])\ \{c \rightarrowtail p[\vec{t}/\vec{x}]\}} *$$

WP-recv
$$\frac{c \rightarrowtail ?(\vec{x})\langle v\rangle\{P\};\ p}{\mathsf{WP}\ c.\mathbf{recv}()\ \{w.\ \exists \vec{t}.\ w = v[\vec{t}/\vec{x}] * P[\vec{t}/\vec{x}] * c \rightarrowtail p[\vec{t}/\vec{x}]\}} *$$

WP-close
$$\frac{P * c \rightarrowtail !\mathbf{end}\{P\}}{\mathsf{WP}\ c.\mathbf{close}()\ \{\mathsf{Emp}\}} *$$

WP-wait
$$\frac{c \rightarrowtail ?\mathbf{end}\{P\}}{\mathsf{WP}\ c.\mathbf{wait}()\ \{P\}} *$$

Figure 39: The LinearActris dependent separation protocols and channel rules.

shall see shortly, the linearity of LinearActris is also crucial for deadlock freedom, as this prevents us from dropping the obligation to send a message over a channel (recall, not sending a message means that the receiving end of the channel would block forever).

6.3.2 *Channels and Protocols*

Like Actris, LinearActris uses dependent separation protocols for reasoning about channels. The grammar of protocols is displayed in Figure 39, and their meaning is as follows:

- *Send protocol* $!(\vec{x})\langle v\rangle\{P\}; p$. The variables $\vec{x}$ are binders that scope over v, P, and p, that is, these are functions of $\vec{x}$. During the verification of a send operation, we can instantiate $\vec{x}$ with mathematical values of our choosing, and then v must be equal to the physical value that is sent, P is a separation logic proposition that we transfer to the receiver, and p is the new protocol for the channel.

- *Receive protocol* $?(\vec{x})\langle v\rangle\{P\}; p$. During the verification of a receive operation, we learn that there exists a choice of mathematical values $\vec{x}$ such that the physical value received equals v, P is a separation logic proposition we receive, and p is the new protocol for the channel.

- *Close protocol* **!end**$\{P\}$. During the verification of a close operation, P is a separation logic proposition that we transfer to the other side.

- *Wait protocol* **?end**$\{P\}$. During the verification of a wait operation, P is a separation logic proposition that we receive.

- *Recursive protocol* $\mu\alpha. p$. This is a recursive protocol, where α is a binder that scopes over p. The recursive protocol can be unfolded by replacing α with p. Recursive protocols with parameters are also supported, we give an example of such a protocol in Section 6.3.4.

The weakest precondition rules for channels in Figure 39 work as follows:

- WP-FORK: This rule is used verify a fork operation. The rule states that fork returns a channel c, and that we can choose a protocol p for this channel. We must then verify that the thread that is spawned on the other side, operates with its side of the channel according to the *dual* protocol $\overline{p}$, which is the same as p except that all send and receive operations are swapped.

- WP-SEND: This rule is used verify a send operation. The rule states that if we have channel ownership $c \rightarrowtail !(\vec{x})\langle v\rangle\{P\}; p$, then we can choose an instantiation $\vec{x} := \vec{t}$. The value that we send must equal $v[\vec{t}/\vec{x}]$, and we must give up ownership of the resources described by the proposition $P[\vec{t}/\vec{x}]$. In the postcondition, the channel gets the new protocol $p[\vec{t}/\vec{x}]$.

- WP-recv: This rule is used verify a receive operation. The rule states that if we have channel ownership $c \rightarrowtail ?(\vec{x})\langle v \rangle\{P\}; p$, then we can receive a message. In the postcondition, we learn that there exists an instantiation $\vec{x} := \vec{t}$ such that the value that we receive equals $v[\vec{t}/\vec{x}]$, and we obtain the ownership of the resources described by the proposition $P[\vec{t}/\vec{x}]$. The channel gets the new protocol $p[\vec{t}/\vec{x}]$.

- WP-close: This rule is used verify a close operation. The rule states that if we have channel ownership $c \rightarrowtail !\mathbf{end}\{P\}$, then we can close the channel. We must also provide the proposition P, which is transmitted to the other side.

- WP-wait: This rule is used verify a wait operation. The rule states that if we have channel ownership $c \rightarrowtail ?\mathbf{end}\{P\}$, then we can wait on the channel, and afterwards we obtain P.

DEADLOCK AND LEAK FREEDOM   The rules in Figure 39 are designed to ensure deadlock- and leak freedom. The reader may note that there are no apparent proof obligations for these properties, other than linearity: there are no preconditions that require us to follow a certain lock- or priority order. In Sections 6.4 to 6.6 we will see how the rules in Figure 39 ensure deadlock freedom and leak freedom.

6.3.3 *Subprotocols*

An important feature of Actris are *subprotocols*, analogous to subtyping in type systems. LinearActris also supports subprotocols. The *subprotocol relation* is written $p_1 \sqsubseteq p_2$, and satisfies the rules in Figure 39. The subprotocol relation lets us make the protocol of a channel more specific: we can turn channel ownership $c \rightarrowtail p_1$ into $c \rightarrowtail p_2$, provided that $p_1 \sqsubseteq p_2$. The rules of Figure 39 are general, and imply various special cases, *e.g.*, the rules allow us to instantiate a quantifier in a send protocol:

$$!(n : \mathbb{N})\langle n \rangle; ?\langle n + 1 \rangle; \mathbf{end} \quad \sqsubseteq \quad !\langle 1 \rangle; ?\langle 2 \rangle; \mathbf{end}$$

Dually, we can abstract a quantifier in a receive protocol:

$$?\langle 1 \rangle; !\langle 2 \rangle; \mathbf{end} \quad \sqsubseteq \quad ?(n : \mathbb{N})\langle n \rangle; !\langle n + 1 \rangle; \mathbf{end}$$

We can apply subprotocols deeper inside the protocol using the special case that if $p_1 \sqsubseteq p_2$, then

$$!\langle v \rangle\{P\}; p_1 \quad \sqsubseteq \quad !\langle v \rangle\{P\}; p_2 \quad \text{and} \quad ?\langle v \rangle\{P\}; p_1 \quad \sqsubseteq \quad ?\langle v \rangle\{P\}; p_2$$

We can also use the subprotocol relation to make the propositions that are transferred more specific: if we have a separating implication $P_1 \twoheadrightarrow P_2$, then we can replace the proposition that is transferred:

$$!\langle v \rangle\{P_2\}; p \quad \sqsubseteq \quad !\langle v \rangle\{P_1\}; p \quad \text{and} \quad ?\langle v \rangle\{P_1\}; p \quad \sqsubseteq \quad ?\langle v \rangle\{P_2\}; p$$

Since $p_1 \sqsubseteq p_2$ gives us $(c \rightarrowtail p_1) \twoheadrightarrow (c \rightarrowtail p_2)$, we can use this to subprotocol the channels that are transferred in a higher-order fashion by taking $P_1 := c \rightarrowtail p_1$ and $P_2 := c \rightarrowtail p_2$.

The subprotocol rules provided in Figure 39 are more powerful than these special cases combined. For instance, the rules also allow us to frame away resources from one step to another:

$$!\langle v \rangle\{P\};\ ?\langle w \rangle\{Q\};\ p \quad \sqsubseteq \quad !\langle v \rangle\{P * R\};\ ?\langle w \rangle\{Q * R\};\ p$$
$$?\langle v \rangle\{P * R\};\ !\langle w \rangle\{Q * R\};\ p \quad \sqsubseteq \quad ?\langle v \rangle\{P\};\ !\langle w \rangle\{Q\};\ p$$

6.3.4 *Guarded Recursive Protocols and Choice*

Another important feature of Actris is the ability to construct infinite protocols. With the constructs we have see so far, we can already construct unbounded protocols and verify programs with them, because one can do *well-founded recursion* in the meta-logic (*i.e.,* a Fixpoint definition in Coq): we can define a recursive function that constructs a protocol, and then use that protocol in a program. This way, we can construct a protocol that sends n messages, and then closes the channel, for any n determined by the first message:

$$!(n : \mathbb{N})\langle n \rangle;\ !\langle n - 1 \rangle;\ \cdots !\langle 0 \rangle;\ \textbf{!end}$$

However, this does not allow us to construct truly infinite protocols, such as the protocol that sends increasing natural numbers forever:

$$!(n : \mathbb{N})\langle n \rangle;\ !\langle n + 1 \rangle;\ !\langle n + 2 \rangle;\ \cdots$$

LinearActris allows us to construct such infinite protocols, using *guarded recursion*:

$$p\ n \triangleq !\langle n \rangle;\ p\ (n + 1) \qquad \text{or formally: } p \triangleq \mu\alpha.\ \lambda n.\ !\langle n \rangle;\ \alpha\ (n + 1)$$

This definition is guarded, because the recursive call is guarded by a message send. Note that our notion of guardedness is a bit more flexible than one might expect; the following definition, in which the recursive call occurs inside the resources, is also guarded:

$$p\ n \triangleq !(c : \text{Addr})\langle c \rangle\{c \rightarrowtail p\ (n + 1)\};\ \textbf{!end}$$

Guarded recursion is most useful in combination with *choice*, which we can encode using a quantified protocol. This lets us express "services" that can perform a certain action (such as sending a natural number) forever, but allow the receiver to close the channel:

$$p \triangleq !(n : N)\langle n \rangle;\ ?(b : \text{Bool})\langle b \rangle;\ \textbf{if } b \textbf{ then } (\textbf{!end}) \textbf{ else } p$$

6.4 FROM MULTI-SHOT TO ONE-SHOT CHANNELS

Before discussing the adequacy proof of LinearActris (Sections 6.5 and 6.6), we first reduce multi-shot channels and protocols to single-shot channels and protocols, inspired by the approach of Dardha et al. (2012) for session types and Jacobs et al. (2023) for separation logic.

The reason we encode multi-shot channels in terms of one-shot channels is twofold. First, it is easier to prove adequacy of the one-shot logic, because it is simpler. The ideas required are not fundamentally different, but there are fewer cases to handle. Second, we believe that the encoding of multi-shot channels in terms of one-shot channels showcases the flexibility of LinearActris: the encoding involves mutable references and transmitting channels over channels and creating new threads in a non-trivial way. If one considers the examples of Section 6.2 in light of the encoding, one realizes that a lot is going on at run-time, and one might therefore expect it to be difficult to verify deadlock and leak freedom. The encoding shows that LinearActris is flexible enough to modularly build the multi-shot abstraction in terms of one-shot channels.

6.4.1 *Primitive One-Shot Channels*

The primitive one-shot channels have the following operations:

fork1 ($\lambda c.\ e$)	Fork a new thread for e with one-shot channel c, and return c.
send1 $c\ v$	Send message v over the channel c.
recv1 c	Receive a message over the channel c, and free c.

The **send1** $c\ v$ and **recv1** c operations may only be used once per one-shot channel.

6.4.2 *Primitive One-Shot Logic*

The primitive one-shot channels are governed by simple one-shot protocols, which are defined in Figure 40. A one-shot protocol is either $!\Phi$ or $?\Phi$, where $\Phi \in \mathsf{Val} \to$ aProp is a separation logic predicate that specifies which values are allowed to be transmitted. The dual of $!\Phi$ is $?\Phi$ and *vice versa*. The primitive one-shot channel weakest precondition rules are given in Figure 40. The rules are similar to the rules of LinearActris, except that they are simpler because they do not have to deal with the complexity of multi-shot channels and protocols:

- WP-PRIM-SEND: When we **send1** $c\ v$, we must have channel ownership $c \rightarrowtail_1 !\Phi$, and we must provide resources $\Phi\ v$ to be transmitted. The postcondition is Emp, because the channel ownership is consumed.

One-shot protocols: ⚙

$$p \in \text{Prot} ::= \,!\Phi \mid ?\Phi \qquad \text{where } \Phi \in \text{Val} \to \text{aProp} \qquad \text{(Protocols)}$$

$$c \rightarrowtail_1 p \qquad \text{(Channel points-to)}$$

$$\overline{!\Phi} \triangleq ?\Phi \qquad \overline{?\Phi} \triangleq \,!\Phi \qquad \text{(Dual)}$$

One-shot channel weakest precondition rules: ⚙

WP-PRIM-FORK
$$\frac{\forall c.\, (c \rightarrowtail_1 \overline{p}) \mathbin{-\!\!*} \text{WP } e\, c\, \{\text{Emp}\}}{\text{WP } \textbf{fork1 } e\, \{c.\, c \rightarrowtail_1 p\}} *$$

WP-PRIM-SEND
$$\frac{\Phi\, v \;\; * \;\; c \rightarrowtail_1 \,!\Phi}{\text{WP } \textbf{send1 } c\, v\, \{\text{Emp}\}} *$$

WP-PRIM-RECV
$$\frac{c \rightarrowtail_1 ?\Phi}{\text{WP } \textbf{recv1 } c\, \{v.\, \Phi\, v\}} *$$

Figure 40: The primitive one-shot channel rules.

- WP-PRIM-RECV: When we **recv1** c, we must have channel ownership $c \rightarrowtail_1 ?\Phi$, and we obtain resources $\Phi\, v$ where v is the value that was received. The channel ownership is consumed.

6.4.3 *Encoding of Multi-Shot Channels*

The multi-shot channels from Section 6.3 are implemented in terms of one-shot channels. The implementation is given in Figure 41. A multi-shot channel endpoint is represented as a mutable reference that stores a one-shot channel. When we send a message v on a multi-shot channel, we create a continuation one-shot channel c', and we send the message (c', v) on the one-shot channel that is stored in the mutable reference. The channel c' is then stored in the mutable reference of the sender, to be used for communicating the next message. On the other side, we receive a message (c', v), and we store c' in the receiver's mutable reference, and then return v. The multi-shot channel is closed by doing a final synchronisation on the one-shot channel without creating a continuation channel, and freeing the mutable reference.

We *define* multi-shot protocols in terms of one-shot protocols, as shown in Figure 39. The definition for $?(\vec{x})\langle v \rangle \{P\};\, p$ specifies that there exists an instantiation of the binders $\vec{x}$, such that the message (c, v) is sent over the one-shot channel, which means that the value is specified by v in the protocol. We additionally transmit the resources P, as well as new channel ownership $c \rightarrowtail_1 P$ for the continuation channel at the right protocol. The definition of the send protocol is simply dual. The

Multi-shot imperative channel implementation: ⚙

$$\mathbf{fork}\ e \triangleq \mathbf{ref}\ (\mathbf{fork1}\ (\lambda c.\ e\ (\mathbf{ref}\ c)))$$
$$\ell.\mathbf{send}(v) \triangleq \mathbf{let}\ c = !\,\ell\ \mathbf{in}\ \ell \leftarrow \mathbf{fork1}\ (\lambda c'.\mathbf{send1}\ c\ (c',v))$$
$$\ell.\mathbf{recv}() \triangleq \mathbf{let}\ (c',v) = \mathbf{recv1}\ (!\,\ell)\ \mathbf{in}\ \ell \leftarrow c';\ v$$
$$\ell.\mathbf{close}() \triangleq \mathbf{send1}\ (!\,\ell)\ ();\ \mathbf{free}\ \ell$$
$$\ell.\mathbf{wait}() \triangleq \mathbf{recv1}\ (!\,\ell);\ \mathbf{free}\ \ell$$

Dependent multi-shot protocol definitions: ⚙

$$?(\vec{x})\langle v\rangle\{P\};\ p \triangleq ?(\lambda w.\ \exists \vec{x}, c.\ w = (c,v) * P * c \rightarrowtail_1 p) \qquad \text{(Receive protocol)}$$
$$!(\vec{x})\langle v\rangle\{P\};\ p \triangleq !(\lambda w.\ \exists \vec{x}, c.\ w = (c,v) * P * c \rightarrowtail_1 \overline{p}) \qquad \text{(Send protocol)}$$
$$?\mathbf{end}\{P\} \triangleq ?(\lambda w.w = () * P) \qquad \text{(Wait protocol)}$$
$$!\mathbf{end}\{P\} \triangleq !(\lambda w.w = () * P) \qquad \text{(Close protocol)}$$
$$\ell \rightarrowtail p \triangleq \exists c, q.\ \ell \mapsto c * c \rightarrowtail_1 q * \rhd(q \sqsubseteq p) \qquad \text{(Channel points-to)}$$

Subprotocols: ⚙

$$!\Phi \sqsubseteq !\Psi \triangleq \forall v.\Psi\ v \twoheadrightarrow \Phi\ v \qquad\qquad !\Phi \sqsubseteq ?\Psi \triangleq \mathsf{False}$$
$$?\Phi \sqsubseteq ?\Psi \triangleq \forall v.\Phi\ v \twoheadrightarrow \Psi\ v \qquad\qquad ?\Phi \sqsubseteq !\Psi \triangleq \mathsf{False}$$

Figure 41: Multi-shot channels and protocols in terms of one-shot channels and protocols.

definitions of the close and wait protocols are special cases of the send and receive protocols, as no continuation channel is created.

Finally, multi-shot channel ownership $\ell \rightarrowtail p$ is defined in terms of heap ownership and one-shot channel ownership, as shown in Figure 39. The definition states that the mutable reference ℓ stores a one-shot channel c, and that the one-shot channel has protocol $q \sqsubseteq p$. This means that the multi-shot channels support subprotocols, even though the one-shot channels do not.

6.5 WHY LINEAR ACTRIS IS DEADLOCK FREE: CONNECTIVITY GRAPHS

Now that we have given the rules of the one-shot logic, we cover how it guarantees deadlock- and leak freedom by linearity. We first give the general structure of the adequacy proof, and explain how it uses an invariant that is preserved as the

program executes (Section 6.5.1). We then give an intuition for the principles that the invariant needs to enforce, by going through some faulty examples, and discuss what it is that makes them deadlock/leak, and how the notion of connectivity graphs (Jacobs et al., 2022a) is used (Section 6.5.2). We finally present how we reason about the preservation of the invariant in terms of connectivity graphs (Section 6.5.3). In the next section we will give a more formal presentation of the adequacy proof, including the use of step-indexing to stratify circular definitions (Section 6.6).

6.5.1 *General Approach*

The general approach we take is to define an invariant $I(\vec{e}, h)$, which describes the state of the configuration of threads and heap. The invariant satisfies three properties that together imply adequacy. This approach is similar to the technique of progress and preservation for proving type safety (Wright and Felleisen, 1994; Pierce, 2002; Harper, 2016), but our invariant is defined semantically (in terms of the operational semantics of the language) instead of syntactically (in terms of inductively defined judgments). The first property is that the invariant can be established by the weakest precondition of the program:

Lemma 6.5.1 (Initialization ✿). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *holds, then* $I([e], \emptyset)$ *holds.*

That is, the invariant holds for the initial configuration with one thread e and empty heap. The second property is that the invariant is preserved by the steps of our operational semantics:

Lemma 6.5.2 (Preservation ✿). *If* $I(\vec{e}, h)$ *holds, and* $(\vec{e}, h) \rightsquigarrow (\vec{e'}, h')$, *then* $I(\vec{e'}, h')$ *holds.*

The third property is that the invariant implies the conclusion of the adequacy theorem:

Lemma 6.5.3 (Progress ✿). *If* $I(\vec{e}, h)$ *holds, then either* $(\vec{e}, h)$ *can step, or* $\vec{e}$ *are all values and* $h = \emptyset$.

Together, these three properties imply adequacy, because if we start with the initial configuration, then we can repeatedly apply the preservation theorem to get to a configuration where the invariant holds, after which we can apply the progress theorem to establish adequacy:

Theorem 6.5.4 (Adequacy ✿). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *is holds, and* $([e], \emptyset) \rightsquigarrow^* (\vec{e}, h)$, *then either:*

- $(\vec{e}, h) \rightsquigarrow (\vec{e'}, h')$ *for some* $(\vec{e'}, h')$, *or,*

- $\vec{e}$ *are all values and* $h = \emptyset$.

In addition to this adequacy theorem, our logic also guarantees safety:

Theorem 6.5.5 (Safety ⚙). *If* $\mathsf{Emp} \vdash \mathsf{WP}\ e\ \{\mathsf{Emp}\}$ *is holds, and* $([e], \emptyset) \rightsquigarrow^* (\vec{e}, h)$, *then every thread in* $\vec{e}$ *can either reduce, or is a value, or is blocked on a receive or wait operation.*

The safety theorem is a straightforward consequence of our invariant, so we will not discuss it further. The reader can find the proof in the Coq mechanization. In the next subsection we aim to give an intuition of what the invariant $I(\vec{e}, h)$ looks like, and why it is preserved by the operational semantics of our language.

6.5.2 *The Invariant Properties*

In this section we investigate the properties that we need the invariant to enforce. We do this by considering program examples that *do* deadlock or leak to identify patterns we need to exclude.

Consider the following example:

```
let c₁ = fork (λc₂. ()) in c₁.recv()
```

The forked-off thread does nothing, and the main thread waits for the forked-off thread by attempting to receive a message. The problem is that the forked-off thread does not fulfill its obligation to send a message. To exclude this pattern the invariant must uphold the following property:

> **Channel fulfillment:** Terminated threads must not hold ownership assertions of channels.

Now consider the following type of deadlock, where both sides try to receive:

```
let c₁ = fork1 (λc₂. recv1 c₂) in recv1 c₁
```

To rule out this example, we require that there cannot be two receive assertions $?\Phi$. The invariant enforces this with the following property:

> **Channel duality:** Each channel in the configuration is in one of two states:
>
> 1. There exist two channel ownership assertions $c \rightarrowtail !\Phi$ and $c \rightarrowtail ?\Phi$ for that channel and the channel buffer is empty.
>
> 2. There exist only the receiver assertion $c \rightarrowtail ?\Phi$ and the channel contains a value v that satisfies $\Phi\ v$.

Next, consider the type of deadlock illustrated by the following example:

```
let l = ref 1 in
let c₁ = fork1 (λc₂. l ← c₂) in
recv1 c₁; send1 (! l) 2; free l
```

In this example, the forked-off thread smuggles its own channel back to the main thread by putting it in the reference l. The main thread then attempts to receive, but this will block forever, as the matching send (on $! l$) is performed after the receive. This example is not ruled out by the invariant property above, as the main thread might be holding both channel ownership assertions for c_1 as well as c_2. The invariant prevents this from happening with the following property:

> **Weak channel acyclicity:** No thread can hold ownership over both endpoints of a channel.

This property is yet again not enough to guarantee deadlock freedom. In general, it can be the case that there are several threads that are waiting for each other, and that none of them will ever perform the send that the others are waiting for. Consider the following situation:

Thread 1:	**recv1** c_1; **send1** d_1 2	**Ownership:**	$c_1 \rightarrowtail_1 ?\Phi * d_1 \rightarrowtail_1 !\Psi$
Thread 2:	**recv1** d_2; **send1** c_2 1	**Ownership:**	$d_2 \rightarrowtail_1 ?\Psi * c_2 \rightarrowtail_1 !\Phi$

Here, both threads are waiting for each other, but neither of them will ever perform the send that the other is waiting for. This does not violate the preceding principle, as it could be the case that channel ownership is held as indicated above. In this case, neither thread holds both channel ownership assertions for the same channel, but there is still a deadlock. We therefore generalize the preceding principle by considering the *graph* of channel ownership assertions held by the threads:

> **Channel acyclicity:** There exists a *connectivity graph* of channel ownership assertions, where there is an edge from a thread T to a channel c if T holds a channel ownership assertion $c \rightarrowtail_1 p$. This graph must be *strongly acyclic*.

By the term *strongly acyclic*, we mean that there is at most one path from any node to another, even if one is allowed to follow edges backwards.

LEAKS   The aforementioned properties are enough to rule out the preceding examples, but there are subtle types of deadlocks that can still occur. The last remaining issue is that we have not yet taken into account the fact that we can store ownership assertions in channels, by transferring them via the send operation.

There is thus a danger that we can leak channel ownership assertions circularly into each other, and thus create a cycle of channel ownership assertions. This could cause deadlocks in the same way as the first example in this section: by leaking a send ownership assertion, a send will never happen, and the receiver will block indefinitely.

For this reason, deadlocks are intimately related to *leaks*. It might be tempting to think that linearity alone is enough to rule out leaks, but as we alluded to, this is not the case. Consider what would happen if we had two channel endpoints c_1 and c_2, and do the following:

c_1.**send**(c_2); c_1.**close**();

This program would not deadlock, but it would put the channel c_2 in the buffer of c_1. If c_1 and c_2 turned out to be two endpoints *of the same channel*, then this would be a leak, as the channel would never be freed. We can choose these protocols for c_1 and c_2:

$$c_1 \rightarrowtail \text{!}(P : \text{aProp})\langle v \rangle\{P\}; \text{ !end} \qquad c_2 \rightarrowtail \text{?}(P : \text{aProp})\langle v \rangle\{P\}; \text{ ?end}$$

This protocol allows us to transfer any resource P, including the ownership assertion for c_2. Thus, channel ownership for the channel would be stored inside itself, and we would have a leak. We strengthen our invariant to ensure that there cannot be any cyclic ownership between channels:

Strong channel acyclicity: Consider the logical connectivity graph of channel ownership assertions, where we have the following edges:

- An edge from a thread T to a channel c if T holds a channel ownership assertion $c \rightarrowtail_1 p$.

- An edge from a channel c to a channel c′ if c contains a message with associated channel ownership assertion $c' \rightarrowtail_1 p$.

We have the invariant that this graph is *strongly acyclic*.

Note that channel ownership should not be confused with having a reference to a channel. A thread can have a reference to a channel without having channel ownership for that channel, and a thread can have channel ownership for a channel without having a reference to that channel.

We can now understand deadlocks and leaks in terms of the connectivity graph:

- **Deadlock.** In order for a thread to be able to perform a receive or wait operation, it must have channel ownership for the channel that it is receiving from. Therefore, if we have a deadlock in which threads are blocked on each other in a circular
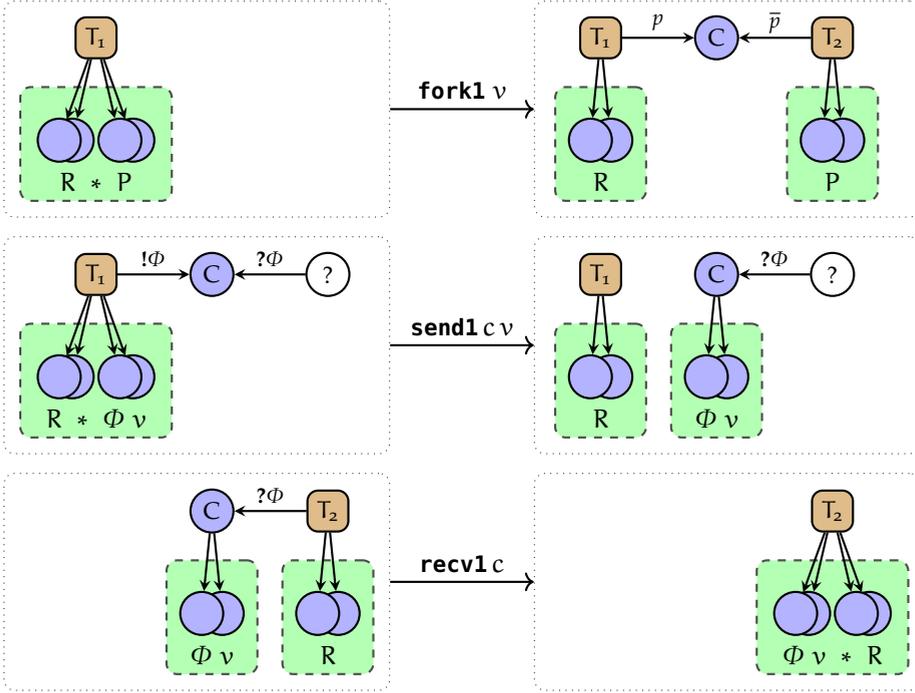
Figure 42: The one-shot channel operations and the corresponding connectivity graph transformations.

manner, then there must be a cycle of threads and channels in the connectivity graph.

- **Leak.** If, after the program has terminated, there are still channels in the heap, then the channel ownership for them must be stored inside each other in a circular manner, and then there must be a cycle of channels in the connectivity graph.

6.5.3 *Preserving the Invariant*

We now discuss how we preserve the invariant by virtue of our program logic rules. The property of channel fulfillment is preserved by the fact that we work in a linear logic. The property of **channel duality** is preserved by the fact that we force channels to be dual when allocated.

The property of **strong channel acyclicity** is more intricate, as the connectivity graph must be updated as the program executes. In Figure 42, we show how the connectivity graph transforms due to each of the one-shot channel operations:

- **Fork.** When thread T_1 does a fork operation, it adds a new thread T_2 to the connectivity graph, and connects it to the original thread via a channel C. The two edges to the channel are labeled with dual protocols p and $\bar{p}$. The original

thread T_1 originally owned separation logic resources R ∗ P, which may contain ownership of other channels (and mutable references, which we ignore here). This is represented as edges from T_1 to the owned channels. We let P be the ownership of the channel ownership that is transferred to the new thread T_2, while R is the part that thread T_1 keeps for itself. Due to this split of ownership, the fork operation corresponds to a modification of the graph, as shown in the figure. Crucially, if the original graph is strongly acyclic, the resulting graph is still strongly acyclic. Note that this relies on the separation between R ∗ P. If we had a channel ownership assertion that occurred both in R and in P, then the resulting graph would not be strongly acyclic.

- **Send.** When thread T_1 performs a send operation on a channel C with protocol !Φ, it must provide resources Φ v, where v is the value it wants to send. The resources Φ v get transferred to the channel, and the thread loses its connection to the channel, because it is one-shot. Therefore, the send operation corresponds to a modification of the graph, as shown in the figure. The reader can see strong acyclicity is preserved.

- **Receive.** When thread T_2 performs a receive operation on a channel C with protocol ?Φ, it receives a value v and resources Φ v from the channel. The channel gets deallocated and removed from the graph, because the channel is one-shot. If the thread initially owned resources R, then afterwards it owns resources Φ v ∗ R. Note that these resources are separated—this relies crucially on the acyclicity of the graph before the receive operation: if thread T_2 already had channel ownership for some channel C′, and additionally got a second channel ownership assertion for C′ via Φ v, then the original graph would not have been strongly acyclic.

In short, the proof rules of LinearActris ensure that strongly acyclic of the connectivity graph is preserved, and thus its adequacy theorem can ensure that the program is deadlock and leak free. In the next section, we will give an overview of how this is proved formally.

6.6   FORMAL ADEQUACY PROOF

In this section we give a formal overview of our adequacy proof. We first give a model of the propositions aProp of LinearActris by solving a recursive domain equation in a step-indexed universe of sets (America and Rutten, 1989; Birkedal et al., 2010) (Section 6.6.1). We then define the invariant that we use in the adequacy proof (Section 6.6.2), and give the semantics of weakest preconditions (Section 6.6.3). Finally, we sketch how the weakest precondition rules and the adequacy theorem are proved (Section 6.6.4).

6.6.1 *The Step-Indexed Model of Propositions*

To map the intuition of the previous section to a formal model of separation logic, we will first give the semantics of the type of propositions. This means we need to define a type aProp with the usual separation logic operators and the connectives $c \rightarrowtail_1 p \in \text{Prot}$ and $\ell \mapsto v \in \text{Prot}$. These connectives assert ownership of outgoing edges to a channel or a heap location in the connectivity graph. To define aProp, we solve the following recursive domain equation:

$$\text{aProp} \simeq (\text{Node} \xrightarrow{\text{fin}} \blacktriangleright \underbrace{(\{!, ?\} \times (\text{Val} \to \text{aProp})}_{\text{protocols } !\Phi \text{ and } ?\Phi} + \underbrace{\text{Val}}_{\text{references}})) \to \text{siProp} \qquad \clubsuit$$

with the overbrace *outgoing edges*.

Here, we let $\text{Node} ::= \text{Thread}(n) \mid \text{Cell}(n)$ be the set of nodes of the connectivity graph, *i.e.,* cells in the heap, which store either a channel or a mutable value, and threads, which are never owned but included for uniformity.

Note that aProp is not well-defined as an inductive or coinductive definition in the category of sets, because the recursive occurrence of aProp is in negative position. That is why we use the results by America and Rutten (1989); Birkedal et al. (2010) to solve the recursive domain equation using step-indexing. The use of step-indexing is evident by the use of (pure) step-indexed propositions siProp as our meta logic, and the use of the $\blacktriangleright$ modality to guard the recursion. This construction is similar to how the model of Iris is constructed, with the crucial difference that Iris considers monotone predicates to obtain an affine logic.

With this definition at hand, we can define the connectives of our separation logic:

$$c \rightarrowtail_1 p \triangleq \lambda\Sigma. \ \Sigma = \{\text{Cell}(c) \mapsto p\} \qquad \clubsuit$$
$$\ell \mapsto v \triangleq \lambda\Sigma. \ \Sigma = \{\text{Cell}(\ell) \mapsto v\}$$
$$P * Q \triangleq \lambda\Sigma. \ \exists\Sigma_1, \Sigma_2. \ \Sigma = \Sigma_1 \cup \Sigma_2 \wedge \text{dom}(\Sigma_1) \cap \text{dom}(\Sigma_2) = \emptyset \wedge P\ \Sigma_1 \wedge Q\ \Sigma_2$$
$$P \mathbin{-\!*} Q \triangleq \lambda\Sigma. \ \forall\Sigma'. \ \text{dom}(\Sigma) \cap \text{dom}(\Sigma') = \emptyset \Rightarrow P\ \Sigma' \Rightarrow Q(\Sigma \cup \Sigma')$$
$$P \wedge Q \triangleq \lambda\Sigma. \ P\ \Sigma \wedge Q\ \Sigma$$

We have glossed over several technical details here, such injections from A into $\blacktriangleright A$, and that the right hand sides of these definitions live in the step-indexed logic siProp. We refer the interested reader to our Coq mechanization for the full details.

6.6.2 *The Invariant*

The invariant is defined in terms of a connectivity graph (Jacobs et al., 2022a), which is a labeled directed graph that is strongly acyclic. The nodes of the graph are the logical objects in the configuration, *i.e.,* threads, channels, and mutable references. The incoming edges of channels are labeled by the protocols $!\Phi$ and $?\Phi$ appearing in

the channel ownership assertions $c \rightarrowtail_1 p$. The incoming edge of a mutable reference is labeled by the value of the reference appearing in the reference ownership assertion $\ell \mapsto v$.

The invariant $I(\sigma)$ on a configuration σ is therefore defined as follows:

$$I(\sigma) \triangleq \exists G : \mathsf{CGraph}. \ \forall v. \ \mathsf{local\_inv}(\sigma[v], \mathsf{in\_labels}_G(v), \mathsf{out\_edges}_G(v)) \qquad \clubsuit$$

where $\mathsf{in\_labels}_G(v)$ is the multiset of labels on incoming edges of v, and $\mathsf{out\_edges}_G(v)$ is a finite map of outgoing edges of v (as in Jacobs et al. (2022a)). Here, $\sigma[v]$ looks up the physical state associated to the logical object v in configuration σ. The value of $\sigma[v]$ is $\mathsf{Expr}(e)$ for a thread, $\mathsf{Ref}(v)$ for a mutable reference containing v, $\mathsf{Chan}(v)$ for a channel containing v, $\mathsf{Chan}(\bot)$ for an empty channel, and $\bot$ if v is not in the configuration at all. The definition of I states that there is a connectivity graph G that is strongly acyclic, and that for every value of v, the local invariant $\mathsf{local\_inv}$ holds. This constrains the relation between the physical state of the object, and the incoming and outgoing edges of the v in the graph, and thus relates the graph to the configuration. The local invariant $\mathsf{local\_inv}$ is defined as follows:

$$\mathsf{local\_inv}(\mathsf{Expr}(e), \alpha, \Sigma) \triangleq \alpha = \emptyset \wedge \mathsf{Emp} \vdash \mathsf{WP}_0 \ e \ \{\mathsf{Emp}\} \ \Sigma \qquad \clubsuit$$
$$\mathsf{local\_inv}(\mathsf{Ref}(v), \alpha, \Sigma) \triangleq \alpha = \{v\} \wedge \Sigma = \emptyset$$
$$\mathsf{local\_inv}(\mathsf{Chan}(v), \alpha, \Sigma) \triangleq \exists \Phi. \ \alpha = \{!\Phi\} \wedge \Phi \ v \ \Sigma$$
$$\mathsf{local\_inv}(\mathsf{Chan}(\bot), \alpha, \Sigma) \triangleq \exists \Phi. \ \alpha = \{!\Phi, ?\Phi\} \wedge \Sigma = \emptyset$$
$$\mathsf{local\_inv}(\bot, \alpha, \Sigma) \triangleq \alpha = \emptyset \wedge \Sigma = \emptyset$$

The local invariant for threads states that the incoming edges are empty, and that we have a WP for the thread expression e, which owns the outgoing edges. The local invariant for references states that the incoming edges are the singleton set containing the value, and that the outgoing edges are empty. The local invariant for a channel that contains a value v states that the incoming edges are the singleton set containing $!\Phi$, and that the outgoing edges are owned by the predicate $\Phi \ v$. The local invariant for an empty channel states that the incoming edges are the set containing $!\Phi$ and $?\Phi$, and that the outgoing edges are empty. The local invariant for a logical object that does not exist in the physical configuration, states that the incoming and outgoing edges are empty.

6.6.3 *Weakest Preconditions*

We have now defined the invariant, but we still need to define the weakest preconditions, which is the main difficulty. In order to do so, we first define a *partial invariant*, which states that the invariant holds for all threads and channels in the configuration, except for the thread that our WP is currently considering: $\clubsuit$

$$I^\circ(\sigma, \text{tid}, \Sigma) \triangleq \exists G : \text{CGraph}. \forall v. \begin{cases} \text{in\_labels}_G(v) = \emptyset \land \text{out\_edges}_G(v) = \Sigma & \text{if } v = \text{tid} \\ \text{local\_inv}(\sigma[v], \text{in\_labels}_G(v), \text{out\_edges}_G(v)) & \text{if } v \neq \text{tid} \end{cases}$$

The partial invariant $I^\circ(\sigma, \text{tid}, \Sigma)$ states that there is a connectivity graph G that is strongly acyclic, and that for every value of v, the local invariant local_inv holds, except for the thread tid, for which we require that the incoming edges are empty, and the outgoing edges are Σ.

Using this partial invariant, we can define the weakest preconditions, which we define by cases depending on whether the expression is a value or not:

$$\text{WP}_\text{o}\ v\ \{\Phi\}\ \Sigma \triangleq \diamond \Phi\ v \qquad\qquad\qquad \text{⚙}$$
$$\text{WP}_\text{o}\ e\ \{\Phi\}\ \Sigma \triangleq \forall \text{tid}, \vec{e}, h. \triangleright I^\circ((\vec{e}, h), \text{tid}, \Sigma) \rightarrow$$
$$\diamond\ (\text{reducible\_or\_blocked}^\circ(e, h, \Sigma) \land \text{preserved}(e, \vec{e}, h, \text{tid}))$$
$$\text{preserved}(e, \vec{e}, h, \text{tid}) \triangleq \forall e', h', \vec{e}_{\text{new}}. (e, h) \rightarrow_\text{p} (e', h', \vec{e}_{\text{new}}) \rightarrow$$
$$\triangleright \exists \Sigma'. I^\circ((\vec{e} + \vec{e}_{\text{new}}, h'), \text{tid}, \Sigma') \land \text{WP}_\text{o}\ e'\ \{\Phi\}\ \Sigma'$$

This definition states that if the expression is a value, then the WP holds if the predicate holds for the value (for technical step-indexing reasons, there is a $\diamond$ modality in front of the predicate, to allow us to remove $\triangleright$ from pure assumptions). If the expression is not a value, then we operate under the assumption that the partial invariant $I^\circ((\vec{e}, h), \text{tid}, \Sigma)$ holds (under the later modality). We must then show that the expression is either reducible or blocked, expressed by the predicate reducible_or_blocked$^\circ(e, h, \Sigma)$. This means that e can either step in the context of the heap h, or that e is blocked on a receive operation on a channel for which Σ contains the $?\Phi$ protocol. Secondly, we must show that the invariant and WP are preserved: if e steps to e', then we must find a Σ' such that the partial invariant holds for the new configuration $(\vec{e} + \vec{e}_{\text{new}}, h')$, and the WP holds for e' under Σ'. Here, $\vec{e}_{\text{new}}$ is the list of new threads that are spawned by the step, and Σ' are the new outgoing edges that are owned by the current thread tid.

RECURSION The reader may have noticed that the definition of the WP and the partial invariant are mutually recursive, in more than one way. This problem is addressed by step-indexing, which allows us to define the WP and the partial invariant using guarded recursion, because all recursive occurrences are under a later modality.

FRAMING For $\text{WP}_\text{o}\ e\ \{\Phi\}$, the frame rule of separation logic does not hold. Inspired by Charguéraud (2020), we can lift it to a *frame preserving* WP, which does satisfy the frame rule:

$$\text{WP}\ e\ \{\Phi\} \triangleq \forall R. \overset{?}{\triangleright} R \ast \text{WP}_\text{o}\ e\ \{v.\ R \ast \Phi\ v\} \qquad\qquad \text{⚙}$$

In this definition, there is a later modality ($\triangleright$) in front of R, but only if e is not a value. This makes sure that we get the *step-framing* rule of Iris: $\triangleright R * WP\ e\ \{\Phi\} \vdash WP\ e\ \{v.\ R * \Phi\ v\}$ if $e \notin \mathsf{Val}$.

6.6.4 *Weakest Precondition Rules and Adequacy*

With the definition of the weakest precondition connective at hand, we prove the weakest preconditions rules of LinearActris. These proofs are relatively complex, as we need to reason about the connectivity graph, and how it is transformed when we perform a step, as shown in Figure 42.

The adequacy proof (Theorem 6.5.4) follows the structure sketched in Section 6.5, by proving the initialization, preservation, and progress theorems. For the progress theorem, we use the fact that the connectivity graph is acyclic, which means that we can always find a thread that can step. Formally, we apply the principle of *waiting induction* (Jacobs et al., 2022a). We refer the interested reader to the Coq mechanization for the full details.

6.7 SEMANTIC TYPING

Semantic type soundness is an approach to proving safety by building a logical relations model. We follow the "logical approach" to semantic typing (Appel et al., 2007; Dreyer et al., 2011; Jung et al., 2018a; Timany et al., 2022) where we define the logical relations model using a program logic—in our case, LinearActris–instead of directly in terms of the operational semantics of the programming language. Our development is based on the semantic type safety proof by Hinrichsen et al. (2021) for an affine session-typed language using the Actris logic. A crucial different is that by using LinearActris instead of Actris, we obtain deadlock- and leak freedom for all typeable programs as a consequence of our strong adequacy theorem (Theorem 6.5.4).

Our type system is inspired by the GV family (Wadler, 2012; Gay and Vasconcelos, 2010), but uses strong updates to track changes to the session types of channels. Moreover, our type system is more expressive than earlier deadlock-free type systems that have appeared in the literature: it supports the combination of session-typed channels with recursive types, subtyping, term- and session type polymorphism, and unique mutable references.

We present the semantic type system and its soundness theorem (Section 6.7.1), and then elaborate on how the semantic type soundness is related to conventional syntactic type soundness (Section 6.7.2).

6.7.1 *Type System*

An overview of the key definitions appears in Figure 43. We omit details about unique mutable references, polymorphism, and copy (a.k.a. unrestricted) types for

Term types:

$$\text{any} \triangleq \lambda w. \, \text{Emp}$$
$$\mathbf{Z} \triangleq \lambda w. \, w \in \mathbb{Z}$$
$$A \multimap B \triangleq \lambda w. \, \forall v. \, \triangleright(A \, v) \twoheadrightarrow \text{wp} \, (w \, v) \, \{B\}$$
$$\text{ch} \, S \triangleq \lambda w. \, w \rightarrowtail S$$

Session types:

$$!A. \, S \triangleq !(v : \text{Val})\langle v \rangle \{A \, v\}; \, S$$
$$?A. \, S \triangleq ?(v : \text{Val})\langle v \rangle \{A \, v\}; \, S$$

Subtyping:

$$A <: B \triangleq \forall v. \, A \, v \twoheadrightarrow B \, v$$
$$S <: T \triangleq S \sqsubseteq T$$

Judgments:

$$\Gamma \vDash \sigma \triangleq \text{\Large ✱}_{(x,A) \in \Gamma} . \, A(\sigma(x))$$
$$\Gamma \vDash e : A \dashv \Gamma' \triangleq \forall \sigma. \, (\Gamma \vDash \sigma) \twoheadrightarrow$$
$$\text{wp} \, e[\sigma] \, \{v. \, A \, v * (\Gamma' \vDash \sigma)\}$$

Semantic typing rules for terms: ⚙

$$\frac{\Gamma_1, x : A \vDash e : B \dashv [\,]}{\Gamma_1 \cdot \Gamma_2 \vDash \lambda x. \, e : A \multimap B \dashv \Gamma_2}$$

$$\frac{\Gamma_1 \vDash e_1 : A \dashv \Gamma_2 \qquad \Gamma_2, x : A \vDash e_2 : B \dashv \Gamma_3}{\Gamma_1 \vDash \mathbf{let} \, x = e_1 \, \mathbf{in} \, e_2 : B \dashv \Gamma_3 \setminus x}$$

Semantic typing rules for channels:

$$\frac{\Gamma_2 \vDash e : \text{ch} \, S \multimap \text{any} \dashv [\,]}{\Gamma_1 \cdot \Gamma_2 \vDash \mathbf{fork} \, e : \text{ch} \, S \dashv \Gamma_1}$$

$$\frac{\Gamma \vDash e : A \dashv \Gamma', x : \text{ch} \, (!A. \, S)}{\Gamma \vDash \mathbf{send} \, x \, e : \mathbf{1} \dashv \Gamma', x : \text{ch} \, S}$$

$$\frac{}{\Gamma, x : \text{ch} \, (?A. \, S) \vDash \mathbf{recv} \, x : A \dashv \Gamma, x : \text{ch} \, S}$$

Figure 43: Typing judgements and type formers of the semantic type system.

brevity's sake, and refer the interested reader to our Coq mechanization and the affine type system which we adopted (Hinrichsen et al., 2021), as the details revolving these aspects are mostly unchanged.

TYPE FORMERS. The type system consists of two kinds of types, term types and session types. We have the usual linear term type constructs such as any, **Z**, and $A \multimap B$, in addition to the channel type ch S, which is parametric on a session type S. We support the usual session types such as !A. S and !A. S, as well as the ones for closing and branching (omitted for brevity's sake).

In a semantic type system, term types are defined as propositions over values (Type $\triangleq$ Val $\to$ aProp). For example, the type ch S is defined in terms of the channel ownership $c \rightarrowtail S$. Session types S are defined using our dependent protocols p. We use the protocol binders to capture that channels exchange values v for which the term type predicate A holds.

TYPING JUDGMENT. As we work with a language with strong updates, we use a typing judgment $\Gamma \vDash e : A \dashv \Gamma'$ with a pre- *and* post-context $\Gamma, \Gamma' \in \text{List}(\text{String} \times \text{Type})$, similar to RustBelt (Jung et al., 2018a). Using the post-context can track how types of variables change throughout evaluation.

We use *closing substitutions* to define our typing contexts, as is standard in logical relation models. Closing substitutions $\sigma \in \text{String} \xrightarrow{\text{fin}} \text{Val}$ are finite partial functions that map the free variables of an expression to corresponding values.

Closing substitutions come with a judgment $\Gamma \vDash \sigma$, which expresses that the closing substitution σ is well-typed in the context Γ. The judgment says that for every typed variable $(x, A) \in \Gamma$ there is a corresponding value in the closing substitution $\sigma(x)$, for which we own the resources $A(\sigma(x))$.

The typing judgment $\Gamma \vDash e : A \dashv \Gamma'$ is defined using our weakest precondition. That is, given a closing substitution σ and resources $\Gamma \vDash \sigma$ for the pre-context Γ, the weakest precondition holds for e (under substitution with σ), with the postcondition stating that the resources $A\,v$ for the resulting value v are owned separately from the resources $\Gamma' \vDash \sigma$ for the post-context Γ'.

TYPING RULES.   In a semantic type system, every typing rule corresponds to a lemma, which states that if the premises hold semantically, then the conclusion holds semantically. These lemmas are proved using the rules of LinearActris, by unfolding the typing judgment and the type formers, which yields goals that are directly provable using the corresponding weakest precondition rules.

SEMANTIC TYPE SOUNDNESS.   As the semantic typing judgment is defined in terms of weakest precondition, we obtain a type soundness theorem as a direct corollary of adequacy (Theorem 6.5.4).

Theorem 6.7.1 (Semantic type soundness ✿). *If* $[\,] \vDash e : \text{any} \dashv [\,]$ *holds, and* $([e], \emptyset) \rightsquigarrow^*$ $(\vec{e}, h)$, *then either* $(\vec{e}, h)$ *can step, or* $\vec{e}$ *are all values and* $h = \emptyset$.

This theorem says that our type systems ensures there are no deadlocks and data leaks. We obtain a similar type soundness theorem for safety (no illegal non-blocking operations, such as use-after-free) using LinearActris's safety theorem (Theorem 6.5.5).

6.7.2 *From Semantic Type Soundness to Syntactic Type Soundness*

Our semantic type system employs the *foundational approach* to logical relations, inspired by Appel and McAllester (2001); Ahmed (2004); Ahmed et al. (2010); Jung et al. (2018a). That is, we define types and type rules semantically as combinators and lemmas in our program logic. The conventional approach to formalizing type systems is to define types syntactically, then define the typing rules as an inductively generated relation, and prove that if a term is syntactically well-typed, it upholds the properties expected by the type system.

We can recover this approach by defining a syntactic type system $\Gamma \vdash e : A \dashv \Gamma'$ whose typing rules mirror the semantic typing rules. This makes it trivial to prove the following theorem:

Theorem 6.7.2 (Fundamental theorem of logical relations). *If a term is syntactically typed, then it is semantically typed, i.e.,* $\Gamma \vdash e : A \dashv \Gamma'$ *implies* $\Gamma \vDash e : A \dashv \Gamma'$.

By combining the fundamental theorem and semantic type soundness, we obtain the conventional syntactic type soundness result:

Corollary 6.7.3 (Syntactic type soundness). *If* $[\,] \vdash e : \text{any} \dashv [\,]$ *is derivable, and* $([e], \emptyset) \rightsquigarrow^* (\vec{e}, h)$, *then either* $(\vec{e}, h)$ *can step, or* $\vec{e}$ *are all values and* $h = \emptyset$.

Because this is trivial yet laborious, we favour the *foundational approach* to semantic typing.

6.8 RELATED AND FUTURE WORK

We first discuss other approaches to prove deadlock freedom (Section 6.8.1), then compare to the original Actris logic (Section 6.8.2), and discuss mechanizations of session types (Section 6.8.3), and channel implementations (Section 6.8.4). Finally, we discuss related work on models of separation logic (Section 6.8.5).

6.8.1 *Proof Methods for Deadlock Freedom*

LINEAR SESSION TYPES   The GV type system (Wadler, 2012; Gay and Vasconcelos, 2010) and follow-up work (Lindley and Morris, 2015, 2016a, 2017; Fowler et al., 2019, 2021) ensure deadlock freedom for a functional language with session types by linearity. Earlier work proved deadlock freedom for a linear π-calculus using a graphical approach (Carbone and Debois, 2010). Toninho et al. (2013); Toninho (2015)'s deadlock-free SILL embeds session-typed processes into a functional language via a monad. Like GV, the seminal paper by Caires and Pfenning (2010) and Toninho (2015)'s PhD thesis spurred a series of derivatives (Caires et al., 2013; Pérez et al., 2014; Das et al., 2018), in which deadlock freedom is guaranteed by linearity. The contribution of our work is to obtain deadlock freedom from linearity in separation logic instead of a type system.

MULTIPARTY SESSION TYPES   Multiparty session types (Honda et al., 2008, 2016) generalize session types from bidirectional channels to n-to-n channels. To ensure deadlock freedom, multiparty session type systems use a consistency check that generalizes the duality condition of binary session types. The consistency check can be performed via projections of a global type, or via an explicit check on a collection of local types (Scalas and Yoshida, 2019). Purely multiparty approaches generally assume a static topology, and thus do not support dynamic creation of threads and channels. This makes them orthogonal in the programs they can establish to be deadlock free compared to linear binary session types (hybrid approaches exist, see below).

LOCK ORDERS   Dijkstra originally proposed lock orders as a mechanism to ensure deadlock freedom for his Dining Philosophers problem (Dijkstra, 1971). Lock orders have been incorporated into a number of verification tools and separation logics

that support proving deadlock freedom, for example (Leino et al., 2010; Le et al., 2013; Zhang et al., 2016; Hamin and Jacobs, 2018). Lock orders are also used in the TaDA Live separation logic for proving liveness of concurrent programs (D'Osualdo et al., 2021b). Lock-order based approaches are orthogonal in expressive strength compared to session types. For instance, it is far from clear how to build a logical relation for a language with session types in terms of a separation logic with lock orders. In the session-typed source language, deadlock freedom is ensured by linearity, and it does not seem possible to translate this into order-based reasoning in the target program logic. Since session types do not have order obligations, it is not clear how the order conditions on the receive operations are justified.

CHOREOGRAPHIES   Choreographic languages (Montesi, 2021; Cruz-Filipe et al., 2021b,a,b) allow the programmer to write a global program that is automatically split into local programs that communicate via channels for which deadlock freedom is guaranteed by construction. Since choreographies are based on program generation, they are very different from our approach.

USAGES AND OBLIGATIONS   Yet another mechanism for deadlock freedom are usages and obligations (Kobayashi, 1997; Igarashi and Kobayashi, 1997; Kobayashi et al., 1999; Igarashi and Kobayashi, 2001; Kobayashi, 2002b; Igarashi and Kobayashi, 2004), which ensure that channels are used in a non-deadlocking order. In contrast to lock orders, the priority involved in usages and obligations always increases in the order. These mechanisms have also been extended to session-typed languages (Dardha and Gay, 2018). Similar to lock orders, usages and obligations entail additional proof obligations, and as such, are orthogonal to obtaining deadlock freedom from linearity.

HYBRID APPROACHES   Message passing has been extended with locks and sharing (Benton, 1994; Villard et al., 2009; Reed, 2009b; Lozes and Villard, 2011, 2012; Pfenning and Griffith, 2015; Balzer et al., 2018, 2019; Hinrichsen et al., 2020; Qian et al., 2021; Rocha and Caires, 2021; Jacobs and Balzer, 2023). Some of these approaches ensure deadlock or leak freedom, *e.g.,* via lock orders, linearity, or other checks. Multiparty session types have been combined with linearity to guarantee progress beyond one session (Carbone et al., 2015, 2016, 2017; Jacobs et al., 2022b). In this chapter we used bidirectional channels (built on top of one-shot channels) as the sole concurrency primitive. In future work, we would like to add locks and multiparty session types, inspired by the preceding work.

6.8.2   *Comparison with Actris*

Separation logic has been extended with session-type based mechanisms to reason about message-passing programs (Francalanza et al., 2011; Lozes and Villard, 2012; Craciun et al., 2015; Oortwijn et al., 2016; Hinrichsen et al., 2020, 2022). Most closely

related to our work is Actris (Hinrichsen et al., 2020, 2022). Our work can be seen as a linear, deadlock-free and leak-free variant of Actris. Actris additionally supports the combination of message passing with locks. Incorporating locks in a deadlock-free manner would be interesting future work.

In contrast to Actris, where channels are implemented using a pair of lock protected buffers, we implement multi-shot channels on top of primitive one-shot channels, inspired by subsequent work on a layered version of Actris (Jacobs et al., 2023). As a result, similar to Jacobs et al. (2023)'s layered version of Actris, we do not support *asynchronous subtyping*, as this is unsound for our implementation of channels. Actris uses a recursive domain equation to constructs multi-shot protocols directly, we follow Jacobs et al. (2023) to construct multi-shot protocols on top of one-shot protocols. We expect that our approach can be applied to a setting with buffered channels and recursive protocols as primitives. An interesting topic for future work would be to investigate if we can obtain deadlock freedom for asynchronous subtyping this way.

Another difference between Actris and LinearActris is that Actris has a single end protocol **end**, transferring no resources, whereas we have two end protocols **!end**{P} and **?end**{P}, which transfer resources. We believe this to be a minor difference, and expect that our results can be extended to Actris with a single **end** protocol.

6.8.3 *Mechanization of Session Types*

Hinrichsen et al. (2021) use Actris to prove soundness of a session type system via the method of semantic typing, inspired by RustBelt (Jung et al., 2018a). We follow a similar approach, but in addition to proving type safety, we prove deadlock and leak freedom. Thiemann (2019) proves type safety of a linear GV-like session type system using dependent types in Agda, Rouvoet et al. (2020) streamline this approach via separation logic. Goto et al. (2016); Ciccone and Padovani (2020); Castro-Perez et al. (2020); Reed (2009a); Chaudhuri et al. (2019) mechanize π-calculus with session types. These works generally show safety, but Jacobs et al. (2022a)'s Coq mechanization shows deadlock freedom. We generalize their approach of connectivity graphs to the context of separation logic. Lastly, Castro-Perez et al. (2021); Jacobs et al. (2022b) mechanize multiparty session types.

6.8.4 *Verification of Message-Passing Implementations*

While channels are a primitive of our operational semantics, others have verified message-passing implementations that use atomic primitives, such as compare-and-swap or atomic-exchange. Mansky et al. (2017) verifies a message-passing system written in C using VST (Appel, 2014; Cao et al., 2018). Tassarotti et al. (2017) proves the correctness of a compiler for an affine session-typed language, showing that the target terminates iff the source program terminates (under fair scheduling

assumptions). In the future, we would like to implement our channels using atomic primitives. In this setting, it is less clear how to formulate the adequacy theorem. As low-level implementations of channels perform busy loops, we would need a notion such as progress under fair scheduling.

Recent work applies Actris to obtain reliable message-passing specifications for channels built on top of UDP-like primitives (Gondelman et al., 2023). Similarly to the shared memory setting, the implementation busy loops until a message has been successfully transferred over the unreliable network, which can only be guaranteed under fair scheduling and a fair network.

6.8.5 *Linear Models of Separation Logic*

The original presentations of sequential separation logic (O'Hearn et al., 2001) and concurrent separation logic (CSL) (O'Hearn, 2004; Brookes, 2004) use a linear model. For sequential separation logic, linearity gives leak freedom, and with scoped CSL-style invariants this scales to concurrent programs that use parallel composition. When extending the language with more general invariants mechanisms that support unscoped thread creation (Hobor et al., 2008; Svendsen and Birkedal, 2014) the situation becomes more complicated. Jung (2020, Thm 2) shows that linearity alone does not give leak freedom, and other mechanisms are needed. (Bizjak et al., 2019)'s Iron logic provides such a mechanism: by disallowing deallocation permissions in invariants, leak freedom can be obtained. Unfortunately, ownership of the **end** protocol needs to include permission to deallocate the channel, making Iron's invariants insufficient for higher-order session types.

While all resources in Iris are affine, and all resources in LinearActris are linear, there have been various efforts to make hybrid models of separation logics that have both linear and affine resources (Tassarotti et al., 2017; Cao et al., 2017; Krebbers et al., 2018; Mansky, 2022). Typically they use some form of partial commutative monoids equipped with an order that specifies which resources can be dropped. The model of LinearActris is an instance of the step-indexed ordered resource algebra model by Krebbers et al. (2018), taking the order to be the reflexive relation, meaning no resources can be dropped. An interesting direction for future work is to add a notion of ghost state to LinearActris, for which these hybrid models could be useful.

We hope that our work can be a step towards bringing deadlock and leak freedom to full-fledged separation logics for fine-grained concurrency, such as Iris and VST. This has been a longstanding challenge, on which recent progress has been made for leak freedom (Bizjak et al., 2019), and termination, as well as termination-preserving refinement (Spies et al., 2021; Tassarotti et al., 2017). Nevertheless, key challenges related to Iris-style invariants remain. As channels can be seen as a particular type of invariant, we hope that our connectivity graph approach can be generalized, *e.g.,* to a linear form of invariants that are compatible with leak- and deadlock freedom.

Part III

PARADOX-FREE PROBABILISTIC PROGRAMMING

Chapter 7

Paradoxes of Probabilistic Programming

ABSTRACT   Probabilistic programming languages allow programmers to write down conditional probability distributions that represent statistical and machine learning models as programs that use observe statements. These programs are run by accumulating likelihood at each observe statement, and using the likelihood to steer random choices and weigh results with inference algorithms such as importance sampling or MCMC. We argue that naive likelihood accumulation does not give desirable semantics and leads to paradoxes when an observe statement is used to condition on a measure-zero event, particularly when the observe statement is executed conditionally on random data. We show that the paradoxes disappear if we explicitly model measure-zero events as a limit of positive measure events, and that we can execute these type of probabilistic programs by accumulating *infinitesimal probabilities* rather than probability densities. Our extension improves probabilistic programming languages as an executable notation for probability distributions by making it more well-behaved and more expressive, by allowing the programmer to be explicit about which limit is intended when conditioning on an event of measure zero.

7.1   INTRODUCTION

Probabilistic programming languages such as Stan (Carpenter et al., 2017), Church (Goodman et al., 2008), and Anglican (Wood et al., 2014) allow programmers to express probabilistic models in statistics and machine learning in a structured way, and run these models with generic inference algorithms such as importance sampling, Metropolis-Hastings, SMC, HMC. At its core, a probabilistic programming language is a notation for probability distributions that looks much like normal programming with calls to random number generators, but with an additional observe construct.

There are two views on probabilistic programming. The pragmatist says that probabilistic programs are a convenient way to write down a likelihood function, and the purist says that probabilistic programs are a notation for structured probabilistic models. The pragmatist interprets an observe statement as "soft conditioning", or imperatively multiplying the likelihood function by some factor. The purist interprets an observe statement as true probabilistic conditioning in the sense of conditional distributions. The pragmatist may also want to write a probabilistic program to compute the likelihood function of a conditional distribution, but the pragmatist is not surprised that there are non-sensical probabilistic programs that do not express

any sensible statistical model. After all, if one writes down an arbitrary likelihood function then it will probably not correspond to a sensible, structured, non-trivial statistical model. The pragmatist blames the programmer for writing non-sensical programs, just as it would have been the fault of the programmer if they had written down the same likelihood function manually. The purist, on the other hand, insists that *any* probabilistic program corresponds to structured statistical model, and that each `observe` statement in a probabilistic program has a probabilistic interpretation whose composition results in the statistical model. We will show that the current state is not satisfactory for the purist, and we will show how to make probabilistic programming languages satisfactory in this respect.

The difficulties with conditioning in probabilistic programs can be traced back to a foundational issue in probability theory. When the event E being conditioned on has nonzero probability, the conditional distribution $\mathbb{P}(A|E)$ is defined as:

$$\mathbb{P}(A|E) = \frac{\mathbb{P}(A \cap E)}{\mathbb{P}(E)}$$

However, this formula for conditional probability is undefined when $\mathbb{P}(E) = 0$, since then also $\mathbb{P}(A \cap E) = 0$ and the fraction $\mathbb{P}(A|E) = \frac{0}{0}$ is undefined. In probabilistic programming we often wish to condition on events E with probability 0, such as "$x = 3.4$", where x is a continuous random variable. There are several methods to condition on measure-zero events. For continuous distributions that have probability density functions, we can replace the probabilities in the above formula with probability densities, which are (usually) nonzero even if $\mathbb{P}(E)$ is zero. For more complicated situations, we can use the Radon–Nikodym derivative or disintegration (Chang and Pollard, 1997; Shan and Ramsey, 2017; Dahlqvist and Kozen, 2020; Ackermann et al., 2017).

A general method for conditioning on measure-zero events is to define a sequence of events E_ϵ parameterized by a number $\epsilon > 0$ such that E_ϵ in some sense converges to E in the limit $\epsilon \to 0$, but $\mathbb{P}(E_\epsilon) > 0$ for all $\epsilon > 0$. We then *define* the conditional distribution to be the limit of $\mathbb{P}(A|E_\epsilon)$:

$$\mathbb{P}(A|E) = \lim_{\epsilon \to 0} \frac{\mathbb{P}(A \cap E_\epsilon)}{\mathbb{P}(E_\epsilon)}$$

In the book *Probability Theory: The Logic of Science (Jaynes, 2003)*, E.T. Jaynes explains that conditioning on measure-zero events is inherently ambiguous, because it depends not just on E but also on the limiting operation E_ϵ we choose:

> Yet although the sequences $\{A_\epsilon\}$ and $\{B_\epsilon\}$ tend to the same limit "$y = 0$", the conditional densities [$\mathbb{P}(x|A_\epsilon)$ and $\mathbb{P}(x|B_\epsilon)$] tend to different limits. As we see from this, merely to specify "$y = 0$" without any qualifications is ambiguous; it tells us to pass to a measure-zero limit, but does not tell us which of any number of limits is intended. [...] Whenever we have a probability density on one space and we wish to generate from it one

on a subspace of measure zero, the only safe procedure is to pass to an explicitly defined limit by a process like [A_ϵ and B_ϵ]. In general, the final result will and must depend on which limiting operation was specified. This is extremely counter-intuitive at first hearing; yet it becomes obvious when the reason for it is understood.

Other methods, such as dividing probability densities, implicitly make the choice E_ϵ for us. Conditioning on events of measure-zero using those methods can lead to paradoxes such as the Borel-Komolgorov paradox, even in the simplest case when probability density functions exist. Paradoxes occur because seemingly unimportant restatements of the problem, such as using a different parameterization for the variables, can affect the choice of E_ϵ that those methods make, and thus change the value of the limit. Consider the following probabilistic program:

```
h = rand(Normal(1.7, 0.5))
if rand(Bernoulli(0.5))
    observe(Normal(h, 0.1), 2.0)
end
```

We first sample a value (say, a person's height) from a prior normally distributed around 1.7 meters and then with probability 0.5 we observe a measurement normally distributed around the height to be 2.0. We ran this program in Anglican with importance sampling, and obtained the following expectation values for h: 1.812 1.814 1.823 1.813 1.806 (10000 samples each). Suppose that we had measured the height in centimeters instead of meters:

```
h = rand(Normal(170, 50))
if rand(Bernoulli(0.5))
    observe(Normal(h, 10), 200)
end
```

We might naively expect this program to produce roughly the same output as the previous program, but multiplied by a factor of 100 to account for the conversion of meters to centimeters. Instead, we get 170.1 170.4 171.5 170.2 169.4. This behavior happens because even though the units of the program appear to be correct, the calculations that importance sampling does to estimate the expectation value involve arithmetic with inconsistent units (in this case, adding a quantity with units m^{-1} to a quantity with neutral units). The issue is not particular to Anglican or importance sampling, but due to the interaction of stochastic branching with way the likelihood is calculated with probability densities; other algorithms (Paige et al., 2014; Tolpin et al., 2015) have the same behavior. In fact, formal semantics based on likelihood accumulation, such as the commutative semantics (Staton, 2017) and the semantics based on on Quasi-Borel spaces (Heunen et al., 2017), also perform arithmetic with

inconsistent units for this example. Lexical likelihood weighting (Wu et al., 2018) does give a unit-consistent answer for this example[1], but still exhibits unit anomalies for other examples described in Section 7.3.

Unit errors in a programming language's implementation or semantics may seem like a very serious issue, but we do not believe that this is a show-stopper in practice, because practitioners can always take the pragmatist view and avoid writing such programs. Although we consider this to be an important foundational issue, it does not invalidate existing work on probabilistic programming.

It is known that conditionals can be problematic. Some inference algorithms, like SMC, will make assumptions that exclude `observe` inside conditionals. For example, (van de Meent et al., 2018) mentions the following when describing SMC:

> Each breakpoint[2] needs to occur at an expression that is evaluated in every execution of a program. In particular, this means that breakpoints should not be associated with expressions inside branches of if expressions. [...] An alternative design, which is often used in practice, is to simply break at every observe and assert that each sample has halted at the same point at run time.

If the design is used where breakpoints happen at every observe, then the assertion that breakpoints should not be associated with expressions inside branches of if expressions will disallow using SMC with programs that have observes inside conditionals. Languages such as Stan, that do not have or do not allow stochastic branching, also do not suffer from the preceding example. In section 7.3 we will show that the problem is not limited to conditionals; there are programs that do not have conditionals but nevertheless have paradoxical behavior. Furthermore, we show that the standard method of likelihood accumulation for implementing probabilistic programming languages can sometimes obtain an answer that disagrees with the purist's exact value for $\mathbb{P}(A|E)$ **even if** $\mathbb{P}(E)$ **is nonzero**, due to a confusion between probabilities and probability densities.

We identify three types of paradoxes that affect probabilistic programming languages that allow dynamically conditioning on events of measure-zero. These paradoxes are based on the idea that it should not matter which parameter scale we use for variables. It shouldn't matter whether we use meters or centimeters to measure height, but it also shouldn't matter whether we use energy density or decibels to measure sound intensity. The change from centimeters to meters

---

1 Many thanks to Alex Lew for pointing this out.

2 A breakpoint in the context of SMC is a designated point in the program where multiple instances of the same program are synchronized and resampled. A simple form of resampling is to take a program instance with current weight w, and flip a coin, and remove the program instance when the coin is heads, and set its weight to $2w$ when the coin is tails. The purpose of this technique is improve efficiency by probabilistically removing program instances that have reached a very low weight, without affecing the overall expecation value. However, resampling is only a good idea if the weight w is small in comparison to other instances, as we would introduce a lot of extra variance otherwise. Breakpoints are used to determine whether a particular weight is small in a relative sense, by comparing it to the weight of other instances at the same breakpoint.

involves a linear parameter transformation by cm = 0.01m, whereas the change from energy density to decibels involves a nonlinear parameter transformation decibels = log(energy density). We give several example programs that show that the output of a probabilistic program can depend on the parameter scale used when we condition on events of measure zero.

Following Jaynes' advice, we extend the language with notation for explicitly choosing *which* limit E_ϵ we mean in an observe statement. We give an implementation of likelihood accumulation using *infinitesimal probabilities* instead of probability densities, and show that this does not suffer from the three types of paradoxes. Infinitesimal probabilities give meaning to conditioning on measure-zero events in terms of a limit of events of strictly positive measure. Since events of strictly positive measure are unproblematic, paradoxes can no longer occur.

Furthermore, we add explicit language support for parameter transformations. This is only soundly possible due to the introduction of infinitesimal probabilities. We show that introducing a parameter transformation in an observe statement does not change the behavior of the probabilistic program. That is, we show that in our language, observe(D,I) has the same behavior as observe(D',I') where D',I' is D,I in a different parameter scale.

Our contributions are the following.

- We identify a problem with existing probabilistic programming languages, in which likelihood accumulation with probability densities can result in three different types of paradoxes when conditioning on a measure-zero event. The three paradoxes violate the principle that the output of a program should not depend on the parameter scale used (Section 7.3).

- We analyze the event that probabilistic programs with observe statements condition on, taking the paradox-free discrete case as a guide, in order to determine what observe ought to mean in the continuous case (Section 7.2).

- We propose a change to probabilistic programming languages to avoid the paradoxes of the continuous measure-zero case, by changing the observe construct to condition on measure-zero events E as an explicit limit $\epsilon \to 0$ of E_ϵ (Sections 7.4 and 7.5), and

  - a method for computing the limit by accumulating *infinitesimal probabilities* instead of probability densities, which we use to implement the adjusted observe construct,

  - a theorem that shows that infinitesimal probabilities correctly compute the limit of E_ϵ, ensuring that programs that use observe on measure-zero events are paradox free,

  - a translation from the existing observe construct to our new observe construct, which gives the same output if the original program was non-paradoxical,

- language support for parameter transformations, which we use to show that the meaning of programs in our language is stable under parameter transformations,

- an implementation of our language as an embedded DSL in Julia (Jacobs, 2020) (Section 7.6).

7.2 ON THE EVENT THAT OBSERVE CONDITIONS ON

Different probabilistic programming languages have different variants of the `observe` statement. Perhaps it's simplest variant, `observe(b)` takes a boolean `b` and conditions on that boolean being true. For instance, if we throw two dice and want to condition on the sum of the dice being 8, we can use this probabilistic program, in pseudocode:

```
function twoDice()
    x = rand(DiscreteUniform(1,6))
    y = rand(DiscreteUniform(1,6))
    observe(x + y == 8)
    return x
end
```

The program `twoDice` represents the conditional distribution $\mathbb{P}(x|x + y = 8)$ where x and y are uniformly distributed numbers from 1 to 6. We wrap the program in a function and use the return value to specify the variable x whose distribution we are interested in. Anglican has a `defquery` construct analogous to the function definition that we use here.

Probabilistic programming languages allow us to sample from the distribution specified by the probabilistic program and compute expectation values. The simplest method to implement `observe` is *rejection sampling* (von Neumann, 1951; Goodman et al., 2008): we start a *trial* by running the program from the beginning, drawing random samples with `rand`, and upon encountering `observe(x + y == 8)` we test the condition, and if the condition is not satisfied we reject the current trial and restart the program from the beginning hoping for better luck next time. If all observes in a trial are satisfied, then we reach the return statement and obtain a sample for x. We estimate expectation values by averaging multiple samples.

What makes probabilistic programs such an expressive notation for probability distributions is that we have access to use the full power of a programming language, such as its control flow and higher order functions (Heunen et al., 2017). The following example generates two random dice throws x and y, and a random boolean b, and uses an observe statement to condition on the sum of the dice throws being 8 if `b = true`, with control flow:

```
x = rand(DiscreteUniform(1,6))
```

```
y = rand(DiscreteUniform(1,6))
b = rand(Bernoulli(0.5))
if b
    observe(x + y == 8)
end
return x
```

This code expresses the conditional probability distribution $\mathbb{P}(x|E)$ where x, y, b are distributed according to the given distributions, and E is the event ($b = true \wedge x + y = 8) \vee (b = false$). That is, a trial is successful if $x + y = 8$ or if $b = false$.

In general, a probabilistic program conditions on the event that the tests of all `observe` statements that are executed succeed. A bit more formally, we have an underlying probability space Ω and we think of an element $\omega \in \Omega$ as the "random seed" that determines the outcome of all `rand` calls (it is sufficient to take $\Omega = \mathbb{R}$; a real number contains an infinite amount of information, sufficient to determine the outcome of an arbitrary number of `rand` calls, even if those calls are sampling from continuous distributions). The execution trace of the program is completely determined by the choice $\omega \in \Omega$. For some subset $E \subset \Omega$, the tests of all the `observe` calls that are executed in the trace succeed. This is the event E that a probabilistic program conditions on. Rejection sampling gives an intuitive semantics for the `observe` statement:

> For a boolean b, the statement `observe(b)` means that we only continue with the current trial only if b = `true`. If b = `false` we reject the current trial.

Unfortunately, rejection sampling can be highly inefficient when used to run a probabilistic program. If we use 1000-sided dice instead of 6-sided dice, the probability that the sum $x + y$ is a particular fixed value is very small, so most trials will be rejected and it may take a long time to obtain a successful sample. Probabilistic programming languages therefore have a construct `observe(D,x)` that means `observe(rand(D) == x)`, but can be handled by more efficient methods such as importance sampling or Markov Chain Monte Carlo (MCMC). The previous example can be written using this type of `observe` as follows:

```
x = rand(DiscreteUniform(1,6))
b = rand(Bernoulli(0.5))
if b
    observe(DiscreteUniform(1,6), 8 - x)
end
return x
```

This relies on the fact that x + y == 8 is equivalent to y == 8 - x. The intuitive semantics of `observe(D,x)` is as follows:

> For discrete distributions D, the statement `observe(D,x)` means that we sample from D and only continue with the current trial if the sampled value is equal to x.

This variant of `observe` can be implemented more efficiently than rejection sampling. We keep track of the `weight` of the current trial that represents the probability that the trial is still active (i.e. the probability that it was not yet rejected). An `observe(D,x)` statement will multiply the `weight` of the current trial by the probability P(D,x) that a sample from D is equal to x:

> For discrete distributions D, the statement `observe(D,x)` gets executed as `weight *= P(D,x)`, where P(D,x) is the probability of x in D.

The output of a trial of a probabilistic program is now a weighted sample: a pair of random value x and a `weight`. Weighted samples can be used to compute expectation values as weighted averages (this is called *importance sampling* [3]). Estimating an expectation value using importance sampling will usually converge faster than rejection sampling with a good proposal distribution, because importance sampling's `observe` will deterministically weigh the trial by the probability P(D,x) rather than randomly rejecting the trial with probability 1 - P(D,x). If P(D,x) = 0.01 then rejection sampling would reject 99% of trials, which is obviously very inefficient. It is important to note that multiplying `weight *= P(D,x)` is the optimized *implementation* of `observe`, and we may still semantically think of it as rejecting the trial if `sample(D) != x`.

If the distribution D is a continuous distribution, then the probability that a sample from D is equal to any particular value x becomes zero, so rejection sampling will reject 100% of trials; it becomes infinitely inefficient. This is not surprising, because on the probability theory side, the event E that we are now conditioning on has measure zero. Importance sampling, on the other hand, continues to work in some cases, provided we replace probabilities with probability densities:

> For continuous distributions D, the statement `observe(D,x)` gets executed as `weight *= pdf(D,x)`, where pdf(D,x) is the probability density of x in D.

---

[3] More advanced MCMC methods can use the weight to make intelligent choices for what to return from `rand` calls, whereas importance sampling uses a random number generator for `rand` calls. We focus on importance sampling because this is the simplest method beyond rejection sampling.

For instance, if we want to compute $\mathbb{E}[x|x + y = 8]$ where x and y are distributed according to $\mathrm{Normal}(2, 3)$ distributions, conditioned on their sum being 8, we can use the following probabilistic program:

```
x = rand(Normal(2,3))
observe(Normal(2,3), 8 - x)
return x
```

This allows us to draw (weighted) samples from the distribution $\mathbb{P}(x|x + y = 8)$ where x, y are distributed according to $\mathrm{Normal}(2, 3)$. Unfortunately, as we shall see in the next section, unlike the discrete case, we do not in general have a probabilistic interpretation for `observe(D,x)` on continuous distributions `D` when control flow is involved, and we can get paradoxical behavior even if control flow is not involved.

7.3 THREE TYPES OF PARADOXES

We identify three types of paradoxes. The first two involve control flow where we either execute observe on different variables in different control flow paths, or an altogether different number of observes in different control flow paths. The third paradox is a variant of the Borel-Komolgorov paradox and involves non-linear parameter transformations.

7.3.1 *Paradox of Type 1: Different Variables Observed in Different Control Flow Paths*

Consider the following probabilistic program:

```
h = rand(Normal(1.7, 0.5))
w = rand(Normal(70, 30))
if rand(Bernoulli(0.5))
   observe(Normal(h, 0.1), 2.0)
else
   observe(Normal(w, 5), 90)
end
bmi = w / h^2
```

We sample a person's height h and weight w from a prior, and then we observe a measurement of the height or weight depending on the outcome of a coin flip. Finally, we calculate the BMI, and want to compute its average. If h' is the measurement sampled from $\mathrm{Normal}(h, 0.1)$ and w' is the measurement sampled from $\mathrm{Normal}(w, 5)$ and b is the boolean sampled from $\mathrm{Bernoulli}(0.5)$, then the event that this program conditions on is $(b = \mathrm{true} \wedge h' = 2.0) \vee (b = \mathrm{false} \wedge w' = 90)$. This event has measure zero.

Just like the program in the introduction, this program exhibits surprising behavior when we change h from meters to centimeters: even after adjusting the formula $bmi = w/(0.01 \cdot h)^2$ to account for the change of units, the estimated expectation value for *bmi* still changes. Why does this happen?

The call to observe(D,x) is implemented as multiplying the weight by the probability density of x in D. Importance sampling runs the program many times, and calculates the estimate for bmi as a weighted average. Thus the program above effectively gets translated as follows by the implementation:

```
weight = 1
h = rand(Normal(1.7, 0.5))
w = rand(Normal(70, 30))
if rand(Bernoulli(0.5))
    weight *= pdf(Normal(h, 0.1), 2.0)
else
    weight *= pdf(Normal(w, 90), 5)
end
bmi = w / h^2
```

Where $\text{pdf}(\text{Normal}(\mu, \sigma), x)$ is the probability density function of the normal distribution:

$$\text{pdf}(\text{Normal}(\mu, \sigma), x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Importance sampling runs this program N times, obtaining a sequence $(bmi_k, weight_k)_{k \in \{1, \ldots, N\}}$.
It estimates $\mathbb{E}[bmi]$ with a weighted average:

$$\mathbb{E}[bmi] \approx \frac{\sum_{k=1}^{N} (weight_k) \cdot (bmi_k)}{\sum_{k=1}^{N} (weight_k)}$$

The problem that causes this estimate to change if we change the units of h is that the formula adds quantities with inconsistent units: some $weight_k$ have unit m^{-1} (inverse length) and some have unit kg^{-1} (inverse mass).

It might be surprising that the weights have units at all, but consider that if we have a probability distribution D over values of unit U, then the probability density function $\text{pdf}(D, x)$ has units U^{-1}. The formula for $\text{pdf}(\text{Normal}(\mu, \sigma), x)$ shows this in the factor of $\frac{1}{\sigma}$ in front of the (unitless) exponential, which has a unit because σ has a unit.

The call pdf(Normal(h, 0.1), 2.0) has units m^{-1} and the call pdf(Normal(w, 90), 5) has units kg^{-1}, and thus the variable weight has units m^{-1} or kg^{-1} depending on the outcome of the coin flip. The weighted average estimate for $\mathbb{E}[bmi]$ adds weights of different runs together, which means that it adds values

of unit m^{-1} to values of unit kg^{-1} . This manifests itself in the estimate changing depending on whether we use m or cm: computations that do arithmetic with inconsistent units may give different results depending on the units used. This calls into question whether this estimate is meaningful, since the estimate depends on whether we measure a value in m or cm, or in kg or g, which arguably should not matter at all.

The reader might now object that conditionally executed `observe` statements are always wrong, and probabilistic programs that use them should be rejected as erroneous. However, in the discrete case there are no unit errors, because in that case the weight gets multiplied by a *probability* rather than a *probability density*, and probabilities are unitless. Furthermore, in the preceding section we have seen that conditionally executed observe statements have a rejection sampling interpretation in the discrete case. This gives the programs a probabilistic meaning in terms of conditional distributions, even if the discrete observe statements are inside conditionals. The event E that is being conditioned on involves the boolean conditions of the control flow. Ideally we would therefore not want to blame the programmer for using conditionals, but change the implementation of `observe` on continuous variables so that the program is meaningful in the same way that the analogous program on discrete variables is meaningful.

7.3.2 *Paradox of Type 2: Different Number of Observes in Different Control Flow Paths*

Let us analyze the program from the introduction:

```
h = rand(Normal(1.7, 0.5))
if rand(Bernoulli(0.5))
    observe(Normal(h, 0.1), 2.0)
end
return h
```

This program exhibits unit anomalies for the same reason: some of the $weight_k$ have units m^{-1} and some have no units, and adding those leads to the surprising behavior. Rather than taking this behavior as a given, let us analyze what this program *ought* to do, if we reason by analogy to the discrete case.

This program has the same structure as the dice program from section 2, the difference being that we now use a normal distribution instead of a discrete uniform distribution. By analogy to that discrete case, the event that is being conditioned on is $(b = \text{true} \wedge h' = 2.0) \vee (b = \text{false})$, where h' is the measurement from $\text{Normal}(h, 0.1)$.

Surprisingly, this event *does not have measure zero* with respect to the Lebesgue measure on $\mathbb{R} \times \mathbb{R} \times \{\text{true}, \text{false}\}$. The event $(b = \text{true} \vee h' = 2.0)$ has measure zero, but the event $b = \text{false}$ has measure $\frac{1}{2}$, so the entire event has measure $\frac{1}{2}$. We can therefore unambiguously apply the definition of conditional probability $\mathbb{P}(A|E) = \frac{\mathbb{P}(A \cap E)}{\mathbb{P}(E)}$. Our probability space is $\Omega = \mathbb{R} \times \mathbb{R} \times \text{bool}$, corresponding to

$h \sim \text{Normal}(1.7, 0.5)$, $h' \sim \text{Normal}(h, 0.1)$, $b \sim \text{Bernoulli}(0.5)$, and $A \subseteq \Omega$ and $E = \{(h, h', b) | (b = \text{true} \wedge h' = 2.0) \vee (b = \text{false})\} \subseteq X$. The posterior $\mathbb{P}(A|E) = \frac{\mathbb{P}(A \cap E)}{\mathbb{P}(E)} = 2 \cdot \mathbb{P}(A \cap E) = 2 \cdot \mathbb{P}(A \cap \{(h, h', b) | b = \text{false}\})$, so the marginal posterior for h is simply $\text{Normal}(1.7, 0.5)$. That is, the whole if statement with the `observe` ought to have no effect.

We can understand this intuitively in terms of rejection sampling: if the sampled boolean $b = \text{true}$, then the observe statement will reject the current trial with probability 1, because the probability of sampling exactly 2.0 from a normal distribution is zero. Hence if $b = \text{true}$ then the trial will almost surely get rejected, whereas if $b = \text{false}$ the trial will not get rejected. The trials where $b = \text{true} \wedge h' = 2.0$ are negligibly rare, so even though the expectation of h is affected *in those trials*, they do not contribute to the final expectation value; only trials with $b = \text{false}$ do.

As an aside: if we added an extra *unconditional* `observe(Normal(h, 0.1), 1.9)` to the program, then the whole event will have measure zero, but nevertheless, trials with $b = \text{false}$ will dominate over trials with $b = \text{true}$, relatively speaking. In general, the control flow path with the least number of continuous observes dominates. If there are multiple control flow paths with minimal number of observes, but also control flow paths with a larger number of observes, we may have a paradox of mixed type 1 & 2.

This reasoning would imply that the if statement and the observe statement are irrelevant; the program ought to be equivalent to `return rand(Normal(1.7, 0.5))`. If this still seems strange, consider the following discrete analogue:

```
h = rand(Binomial(10000, 0.5))
if rand(Bernoulli(0.5))
    observe(binomial(10000, 0.9), h)
end
return h
```

That is, we first sample h between 0 and 10000 according to a binomial distribution, and then with probability 0.5 we observe that h is equal to a number sampled from another binomial distribution that gives a number between 0 and 10000. Since that binomial distribution is highly biased toward numbers close to 10000, we might expect the average value of h to lie significantly higher than 5000. This is not the case. The rejection sampling interpretation tells us that most of the trials where the coin flipped true, will be rejected, because the sample from $\text{Binomial}(10000, 0.9)$ is almost never equal to h. Thus, although *those* samples have an average significantly above 5000, almost all of the successful trials will be trials where the coin flipped false, and thus the expected value of h will lie very close to 5000.

Since we know that rejection sampling agrees with importance sampling in expectation, importance sampling will also compute an estimate for the expectation value of h that lies very close to 5000. The further we increase the number 10000, the stronger this effect becomes, because the probability that the second sample is

equal to h further decreases. In the continuous case this probability becomes 0, so the successful samples will almost surely be from trials where the coin flipped to false. Therefore the average value of h in the continuous case should indeed be 170, unaffected by the if statement and the observe.

Another way to express this point, is that in the discrete case importance sampling, rejection sampling, and the exact value given by the conditional expectation are all in agreement, even if conditionals are involved. On the other hand, in the continuous case, importance sampling with probability densities gives a different answer than rejection sampling and the exact value given by the conditional expectation $\mathbb{E}[h|E]$ (the latter two *are* equal to each other; both 1.7).

The reader may insist that the semantics of the program is *defined* to be weight accumulation with probability densities, that is, the semantics of the program is *defined* to correspond to

```
weight = 1
h = rand(Normal(1.7, 0.5))
if rand(Bernoulli(0.5))
    weight *= pdf(Normal(h, 0.1), 2.0)
end
return h
```

We can only appeal to external principles to argue against this, such as unit consistency, analogy with the discrete case, the probabilistic interpretation of observe, and the rejection sampling interpretation of observe, but the reader may choose to lay those principles aside and take this *implementation* of observe as the *semantics* of observe. We do hope to eventually convince this reader that a *different* implementation of observe that does abide by these principles, could be interesting. Although our semantics will differ from the standard one, it will agree with lexicographic likelihood weighting (Wu et al., 2018) for this example, which does not exhibit this particular paradox.

7.3.3 *Paradox of Type 3: Non-Linear Parameter Transformations*

Consider the problem of conditioning on $x = y$ given $x \sim \text{Normal}(10, 5)$ and $y \sim \text{Normal}(15, 5)$, and computing the expectation $\mathbb{E}[\exp(x)]$. Written as a probabilistic program,

```
x = rand(Normal(10,5))
observe(Normal(15,5),x)
return exp(x)
```

In a physical situation, x, y might be values measured in decibels and $\exp(x), \exp(y)$ may be (relative) energy density. We could change parameters to $a = \exp(x)$ and

$b = \exp(y)$. Then $a \sim \mathrm{LogNormal}(10, 5)$ and $b \sim \mathrm{LogNormal}(15, 5)$. Since the event $x = y$ is the same as $\exp(x) = \exp(y)$, we might naively expect the program to be equivalent to:

```
a = rand(LogNormal(10,5))
observe(LogNormal(15,5),a)
return a
```

This is not the case. The two programs give different expectation values. Compared to type 1 & 2 paradoxes, this type 3 paradox shows that the subtlety is not restricted to programs that have control flow or to distributions that are not continuous; the normal and lognormal distributions are perfectly smooth.

This paradox is closely related to the Borel-Komolgorov paradox. Another variant of the original Borel-Komolgorov paradox has been expressed in Hakaru (Shan and Ramsey, 2017), but is not expressible in Anglican or Stan. The reason is that Hakaru allows the programmer to condition a measure-zero condition $f(x, y) = 0$ such as $x + y - 8 = 0$ directly without having to manually invert the relationship to $y = 8 - x$, and performs symbolic manipulation to do exact Bayesian inference. Hakaru allows a single such observe at the very end of a program, which allows it to sidestep the previous paradoxes related to control flow. The semantics of the single observe is defined by disintegration, which means that the semantics of a Hakaru program depends on the form of f. That is, if we take another function g with the same solution set $g(x, y) = 0$ as f, the output may change. The programmer can use this mechanism to control which event they want to condition on. Our version of the paradox shows that the subtlety of conditioning on measure-zero events is not restricted to programs that use that type of disintegration.

7.4 AVOIDING EVENTS OF MEASURE ZERO WITH INTERVALS

Unit anomalies cannot occur with discrete distributions, because in the discrete case we only deal with probabilities and not with probability densities. Recall that for discrete probability distributions `D`, an observe statement `observe(D,x)` gets executed as `weight *= P(D,x)` where `P(D,x)` is the probability of `x` in the distribution `D`. Probabilities have no units, so the `weight` variable stays unitless and the weighted average is always unit correct if the probabilistic program is unit correct, even if `observe` statements get executed conditionally. Furthermore, in the discrete case we have a probabilistic and rejection sampling interpretation of observe, and we may view weight accumulation as an optimization to compute the same expectation values as rejection sampling, but more efficiently. We wish to extend these good properties to the continuous case.

The reason that the discrete case causes no trouble is not due to `D` being discrete per se. The reason it causes no trouble is that `P(D,x)` is a probability rather than a probability density. In the continuous case the probability that `rand(D) == x` is

zero, and that's why it was necessary to use probability densities. However, even in the continuous case, the probability that a sample from `D` lies in some *interval* is generally nonzero. We shall therefore change the observe statement to `observe(D,I)` where `I` is an interval, which conditions on the event rand(D) $\in$ I. In the discrete case we can allow `I` to be a singleton set, but in the continuous case we insist that `I` is an interval of nonzero width.

We have the following rejection sampling interpretation for `observe(D,I)`:

> For continuous or discrete distributions `D`, the statement `observe(D,I)` means that we sample from `D` and only continue with the current trial if the sampled value lies in `I`.

And the following operational semantics for `observe(D,I)`:

> For continuous or discrete distributions `D`, the statement `observe(D,I)` gets executed as `weight *= P(D,I)` where `P(D,I)` is the probability that a value sampled from `D` lies in `I`.

Let $I = [a, b] = \{x \in \mathbb{R} : a \leqslant x \leqslant b\}$. We can calculate $\mathbb{P}(\text{rand}(D) \in [a, b]) = \text{cdf}(D, b) - \text{cdf}(D, a)$ using the cumulative density function $\text{cdf}(D, x)$. This probability allows us to update the `weight` of the trial. For instance, a call `observe(Normal(2.0,0.1),[a,b])` is executed as `weight *= normalcdf(2.0,0.1,b)` `- normalcdf(2.0,0.1,a)` where $\text{normalcdf}(\mu, \sigma, x)$ is the cumulative density function for the normal distribution.

Notice how this change from probability densities to probabilities prevents unit anomalies: if we change the variables a, b from meters to centimeters, then we must write `observe(Normal(200,10), [a,b])`, which gets executed as `weight *= normalcdf(200,10,b) - normalcdf(200,10,a)`. We introduced a factor 100 to convert μ and σ from meters to centimeters. This conversion ensures that the result of the program remains unchanged, because $\text{normalcdf}(r\mu, r\sigma, rx) = \text{normalcdf}(\mu, \sigma, x)$ for all $r > 0$. Hence the computed `weight` will be exactly the same whether we work with meters or centimeters. On the other hand, for the probability density function it is **not** the case that $\text{normalpdf}(r\mu, r\sigma, rx) = \text{normalpdf}(\mu, \sigma, x)$. It is precisely this lack of invariance that causes unit anomalies with probability densities.

7.4.1 *Conditioning on Measure Zero Events as a Limit of Positive Measure Events*

We can approximate the old `observe(D,x)` behavior with `observe(D,I)` by choosing $I = [x - \frac{1}{2}w, x + \frac{1}{2}w]$ to be a very small interval of width `w` around `x` (taking `w` to be a small number, such as `w = 0.0001`). This has two important advantages over `observe(D,x)`:

1. We no longer get unit anomalies or other paradoxes; if we change the units of x, we must also change the units of w, which keeps the weight the same.

2. Unlike for observe(D,x), we have an unambiguous probabilistic and rejection sampling interpretation of observe(D,I) for intervals of nonzero width, because the event being conditioned on has nonzero measure.

However, the number w = 0.0001 is rather arbitrary. We would like to let $w \rightarrow 0$ and recover the functionality of observe(D,x) to condition on an exact value. With sufficiently small w we can get arbitrarily close, but we can never recover its behavior exactly.

We therefore parameterize probabilistic programs by a dimensionless parameter eps. The BMI example then becomes:

```
function bmi_example(eps)
   h = rand(Normal(170, 50))
   w = rand(Normal(70, 30))
   if rand(Bernoulli(0.5))
     observe(Normal(200, 10), (h, A*eps))
   else
      observe(Normal(90, 5), (w, B*eps))
   end
   return w / h^2
end
```

Since eps is dimensionless, we can not simply use eps as the width of the intervals: because h is in cm, the width of the interval around h has to be in cm, and the width of the interval around w has to be in kg. We are forced to introduce a constant A with units cm and a constant B with units kg that multiply eps in the widths of the intervals in the observes.

We could now run importance sampling on bmi_example(eps) for n=10000 trials for eps=0.1, eps=0.01, eps=0.001 and so on, to see what value it converges to. If we run each of these independently, then the rand calls will give different results, so there will be different randomness in each of these, and it may be difficult to see the convergence. In order to address this, we can run the program with different values of eps but with the same random seed for the random number generator. This will make the outcomes of the rand calls the same regardless of the value of eps. In fact, for a given random seed, the result of running importance sampling for a given number of trials will be a deterministic function f(seed,eps) of the random seed and eps

If we assume that the program uses eps = ϵ only in the widths of the intervals, and not in the rest of the program, then for a fixed seed, the function f(seed, ϵ) will be a function of ϵ of a specific form, because importance sampling computes

$$f(\text{seed}, \epsilon) = \frac{\sum_{k=1}^{N}(weight_k(\epsilon)) \cdot (value_k)}{\sum_{k=1}^{N}(weight_k(\epsilon))}$$

In this fraction, the $weight_k$ are a function of ϵ, but the $value_k$ are independent of ϵ if ϵ only occurs inside the widths of intervals. Since the weight gets multiplied by $P(D, I)$ on each observe(D,I), the $weight_k(\epsilon)$ is of a very specific form:

$$weight_k(\epsilon) = C \cdot P(D_1, (x_1, w_1\epsilon)) \cdots P(D_n, (x_n, w_n\epsilon))$$

where the constant C contains all the probabilities accumulated from observes that did not involve ϵ, multiplied by a product of probabilities that did involve ϵ. Since $P(D, (x, w\epsilon)) = \text{cdf}(D, x + \frac{1}{2}w\epsilon) + \text{cdf}(D, x - \frac{1}{2}w\epsilon)$, we could, in principle determine the precise function $weight_k(\epsilon)$ and hence f(seed, ϵ) for any given seed. We could then, in principle, compute the exact limit of this function as $\epsilon \to 0$, with a computer algebra system. This is, of course, impractical. The next section shows that we can compute the limit efficiently by doing arithmetic with infinitesimal numbers.

7.5 USING INFINITESIMAL NUMBERS FOR MEASURE-ZERO OBSERVATIONS

In order to recover the behavior of the old observe(D,x) using observe(D,I) with an interval $I = [x - \frac{1}{2}w, x + \frac{1}{2}w]$, we want to take the limit $w \to 0$, to make $[x - \frac{1}{2}w, x + \frac{1}{2}w]$ an infinitesimally small interval around x. We accomplish this using symbolic infinitesimal numbers[4] of the form $r\epsilon^n$, where $r \in \mathbb{R}$ and $n \in \mathbb{Z}$. We allow $n < 0$, so that $r\epsilon^n$ can also represent "infinitely large" numbers as well as "infinitesimally small" numbers. We will not make use of this possibility, but it makes the definitions and proofs more general and more uniform.[5]

Definition 7.5.1. An infinitesimal number is a pair $(r, n) \in \mathbb{R} \times \mathbb{Z}$, which we write as $r\epsilon^n$.[6]

The infinitesimals of the form $r\epsilon^0$ correspond to the real numbers.

---

4 In the philosophy literature there has been work on using non-standard analysis and other number systems to handle probability 0 events, see (Pedersen, 2014) and (Hofweber, 2014) and references therein.

5 These infinitesimal numbers may be viewed as the leading terms of Laurent series. This bears some resemblance to the dual numbers used in automatic differentiation, which represent the constant and linear term of the Taylor series. In our case we only have the first nonzero term of the Laurent series, but the order of the term is allowed to vary.

6 The exponent n of ϵ will play the same role as the number of densities d in lexicographic likelihood weighting(Wu et al., 2018).

Definition 7.5.2. Addition, subtraction, multiplication, and division on those infinitesimal numbers are defined as follows:

$$r\epsilon^n \pm s\epsilon^k = \begin{cases} (r \pm s)\epsilon^n & \text{if } n = k \\ r\epsilon^n & \text{if } n < k \\ \pm s\epsilon^k & \text{if } n > k \end{cases}$$

$$(r\epsilon^n) \cdot (s\epsilon^k) = (r \cdot s)\epsilon^{n+k}$$

$$(r\epsilon^n)/(s\epsilon^k) = \begin{cases} (r/s)\epsilon^{n-k} & \text{if } s \neq 0 \\ \text{undefined} & \text{if } s = 0 \end{cases}$$

Like ordinary division, division of infinitesimals is a partial function, which is `undefined` if the denominator is *exactly* zero.

These rules may be intuitively understood by thinking of ϵ as a very small number; e.g. if $n < k$ then ϵ^k will be negligible compared to ϵ^n, which is why we define $r\epsilon^n + s\epsilon^k = r\epsilon^n$ in that case, and keep only the lowest order term.

We represent intervals $[x - \frac{1}{2}w, x + \frac{1}{2}w]$ as midpoint-width pairs (x, w), where w may be an infinitesimal number.

Definition 7.5.3. If D is a continuous distribution, we compute the probability $P(D, (x, w))$ that $X \sim D$ lies in the interval (x, w) as:

$$P(D, (x, w)) = \begin{cases} \text{cdf}(D, x + \frac{1}{2}r) - \text{cdf}(D, x - \frac{1}{2}r) & \text{if } w = r\epsilon^0 \text{ is not infinitesimal} \\ \text{pdf}(D, x) \cdot r\epsilon^n & \text{if } w = r\epsilon^n \text{ is infinitesimal } (n > 0) \end{cases} \tag{2}$$

Where $\text{cdf}(D, x)$ and $\text{pdf}(D, x)$ are the cumulative and probability density functions, respectively.

Note that the two cases agree in the sense that if w is very small, then

$$\text{cdf}(D, x + \frac{1}{2}w) - \text{cdf}(D, x - \frac{1}{2}w) \approx \frac{d}{dx}\text{cdf}(D, x) \cdot w = \text{pdf}(D, x) \cdot w$$

Definition 7.5.4. We say that $f(x)$ is a "probability expression" in the variable x if $f(x)$ is defined using the operations $+, -, \cdot, /$, constants, and $P(D, (s, rx))$ where $r, s \in \mathbb{R}$ are constants, and D is a probability distribution with differentiable cdf.

We can view f as a function from reals to reals (on the domain on which it is defined, that is, excluding points where division by zero happens), or as a function from infinitesimals to infinitesimals by re-interpreting the operations in infinitesimal arithmetic. The value of $f(\epsilon)$ on the symbolic infinitesimal ϵ tells us something about the limiting behavior of $f(x)$ near zero:

Theorem 7.5.5. *If $f(x)$ is a probability expression, and if evaluation of $f(\epsilon)$ is not* `undefined`, *and $f(\epsilon) = r\epsilon^n$, then* $\lim_{x \to 0} \frac{f(x)}{x^n} = r$.

Note that the theorem only tells us that $\lim_{x \to 0} \frac{f(x)}{x^n} = r$ *if* $f(\epsilon)$ evaluates to $r\epsilon^n$ with infinitesimal arithmetic. If evaluating $f(\epsilon)$ results in division by zero, then the theorem does not give any information. In fact, the converse of the theorem does *not* hold: it may be that $\lim_{x \to 0} \frac{f(x)}{x^n} = r$ but evaluating $f(\epsilon)$ results in division by zero.

Proof. By induction on the structure of the expression.
We know that evaluation of $f(\epsilon)$ did not result in division by zero, and $f(\epsilon) = r\epsilon^n$.
We need to show that $\lim_{x \to 0} \frac{f(x)}{x^n} = r$.

- If $f(x)$ is a constant r, then we have $f(\epsilon) = r\epsilon^0$, and indeed $\lim_{x \to 0} \frac{f(x)}{x^0} = \lim_{x \to 0} f(x) = r$.

- If $f(x) = P(D, (s, rx))$. Now $f(\epsilon) = pdf(D, s) \cdot r\epsilon$, and

$$
\begin{aligned}
pdf(D, s) \cdot r &= r\frac{d}{dx}[cdf(D, x)]_{x=s} \\
&= r\lim_{x \to 0} \frac{cdf(D, s + x) - cdf(D, s - x)}{2x} \\
&= \lim_{x' \to 0} \frac{cdf(D, s + \frac{1}{2}rx') - cdf(D, s - \frac{1}{2}rx')}{x'} \\
&= \lim_{x' \to 0} \frac{P(D, (s, rx'))}{x'}
\end{aligned}
$$

- If $f(x) = g(x) + h(x)$. Since evaluation of $f(\epsilon)$ did not result in division by zero, neither did evaluation of the subexpressions $g(\epsilon)$ and $h(\epsilon)$, so $g(\epsilon) = r_1\epsilon^{n_1}$ and $h(\epsilon) = r_2\epsilon^{n_2}$ for some r_1, r_2, n_1, n_2. Therefore, by the induction hypothesis we have $\lim_{x \to 0} \frac{g(x)}{x^{n_1}} = r_1$ and $\lim_{x \to 0} \frac{h(x)}{x^{n_2}} = r_2$.

- Case $n_1 = n_2$: Now $f(\epsilon) = (r_1 + r_2)\epsilon^{n_1}$, and we have

$$
\lim_{x \to 0} \frac{f(x)}{x^{n_1}} = \lim_{x \to 0} \frac{g(x) + h(x)}{x^{n_1}} = \lim_{x \to 0} \frac{g(x)}{x^{n_1}} + \lim_{x \to 0} \frac{h(x)}{x^{n_1}} = r_1 + r_2
$$

- Case $n_1 < n_2$: Now $f(\epsilon) = r_1\epsilon^{n_1}$, and since $\lim_{x \to 0} \frac{h(\epsilon)}{x^{n_2}} = r_2$ we have

$$
0 = 0 \cdot r_2 = (\lim_{x \to 0} x^{n_2 - n_1}) \cdot (\lim_{x \to 0} \frac{h(x)}{x^{n_2}}) = \lim_{x \to 0} \frac{x^{n_2 - n_1} h(x)}{x^{n_2}} = \lim_{x \to 0} \frac{h(x)}{x^{n_1}}
$$

Therefore

$$
\lim_{x \to 0} \frac{f(x)}{x^{n_1}} = \lim_{x \to 0} \frac{g(x) + h(x)}{x^{n_1}} = \lim_{x \to 0} \frac{g(x)}{x^{n_1}} + \lim_{x \to 0} \frac{h(x)}{x^{n_1}} = r_1
$$

- Case $n_1 > n_2$. Analogous to the previous case.

- If $f(x) = g(x) - h(x)$. Analogous to the case for addition.

- If $f(x) = g(x) \cdot h(x)$. Since evaluation of $f(\epsilon)$ did not result in division by zero, neither did evaluation of the subexpressions $g(\epsilon)$ and $h(\epsilon)$, so $g(\epsilon) = r_1 \epsilon^{n_1}$ and $h(\epsilon) = r_2 \epsilon^{n_2}$ for some r_1, r_2, n_1, n_2. Therefore, by the induction hypothesis we have $\lim_{x \to 0} \frac{g(x)}{x^{n_1}} = r_1$ and $\lim_{x \to 0} \frac{h(x)}{x^{n_2}} = r_2$. Then

$$\lim_{x \to 0} \frac{f(x)}{x^{n_1 + n_2}} = \lim_{x \to 0} \frac{g(x)}{x^{n_1}} \cdot \frac{h(x)}{x^{n_2}} = (\lim_{x \to 0} \frac{g(x)}{x^{n_1}}) \cdot (\lim_{x \to 0} \frac{h(x)}{x^{n_2}}) = r_1 \cdot r_2$$

- If $f(x) = g(x)/h(x)$. Since evaluation of $f(\epsilon)$ did not result in division by zero, neither did evaluation of the subexpressions $g(\epsilon)$ and $h(\epsilon)$, so $g(\epsilon) = r_1 \epsilon^{n_1}$ and $h(\epsilon) = r_2 \epsilon^{n_2}$ for some r_1, r_2, n_1, n_2. Therefore, by the induction hypothesis we have $\lim_{x \to 0} \frac{g(x)}{x^{n_1}} = r_1$ and $\lim_{x \to 0} \frac{h(x)}{x^{n_2}} = r_2$. By the assumption that no division by exactly zero occurred in the evaluation of $f(\epsilon)$, we have $r_2 \neq 0$. Then

$$\lim_{x \to 0} \frac{f(x)}{x^{n_1 + n_2}} = \lim_{x \to 0} \frac{g(x)}{x^{n_1}} / \frac{h(x)}{x^{n_2}} = (\lim_{x \to 0} \frac{g(x)}{x^{n_1}}) / (\lim_{x \to 0} \frac{h(x)}{x^{n_2}}) = r_1 / r_2$$

This finishes the proof. $\square$

Some subtleties of limits and infinitesimals

In order to think about infinitesimals one must first choose a function $f(x)$ of which one wishes to learn something about the limit as $x \to 0$. Thinking about infinitesimal arithmetic independent of such a function leads to confusion. Furthermore, the result of evaluating $f(\epsilon)$ depends not just on $f(x)$ as a function on real numbers, but also on the arithmetic expression used for computing f. Consider the functions f, g:

$$f(x) = 5 \cdot x^2 + 0 \cdot x$$
$$g(x) = 5 \cdot x^2$$

As functions on real numbers, $f = g$, but nevertheless, with infinitesimal arithmetic their results differ:

$$f(\epsilon) = 0 \cdot \epsilon^1$$
$$g(\epsilon) = 5 \cdot \epsilon^2$$

Applying the theorem to these results gives the following limits for f and g:

$$\lim_{x \to 0} \frac{f(x)}{x} = 0$$
$$\lim_{x \to 0} \frac{g(x)}{x^2} = 5$$

Both of these limits are correct, but this example shows that *which* limit the theorem says something about may depend on how the function is computed. The limit for g

gives more information than the limit for f; the limit for f is conservative and does not tell us as much as the limit for g does. Fortunately, this won't be a problem for our use case: we intend to apply the theorem to the weighted average of importance sampling, where the probabilities may be infinitesimal numbers. In this case the power of ϵ of the numerator and denominator are always the same, so the final result will always have power ϵ^0, and the theorem will then tell us about the limit $\lim_{x \to 0} \frac{f(x)}{x^0} = \lim_{x \to 0} f(x)$.

Another subtlety is that the converse of the theorem does not hold. It is possible that $\lim_{x \to 0} \frac{f(x)}{x^n} = r$, but evaluation of $f(\epsilon)$ with infinitesimal arithmetic results in division by exactly zero. An example is $f(x) = \frac{x^2}{(x+x^2)-x}$. We have $\lim_{x \to 0} f(x) = 1$, but when evaluating $f(\epsilon) = \frac{\epsilon^2}{(\epsilon+\epsilon^2)-\epsilon}$, division by zero occurs, because we have the evaluation sequence:

$$\frac{\epsilon^2}{(\epsilon + \epsilon^2) - \epsilon} \to \frac{\epsilon^2}{\epsilon - \epsilon} \to \frac{\epsilon^2}{0} \to \text{undefined}$$

If we used full Laurent series $a_k \epsilon^k + a_{k+1} \epsilon^{k+1} + \dots$ as our representation for infinitesimal numbers, then we would potentially be able to compute more limits, even some of those where exact cancellation happens in a denominator. Keeping only the first term is sufficient for our purposes, and more efficient, because our infinitesimal numbers are pairs (r, n) of a real (or floating point) number r and an integer n, whereas Laurent series are infinite sequences of real numbers $(a_k, a_{k+1}, \dots)$.

The lemmas about computing limits have the form "For all $a, b \in \mathbb{R}$, if $\lim_{x \to 0} f(x) = a$, and $\lim_{x \to 0} g(x) = b$, and $b \neq 0$, then $\lim_{x \to 0} \frac{f(x)}{g(x)} = \frac{\lim_{x \to 0} f(x)}{\lim_{x \to 0} g(x)}$". It is *not* true in general that $\lim_{x \to 0} \frac{f(x)}{g(x)} = \frac{\lim_{x \to 0} f(x)}{\lim_{x \to 0} g(x)}$. It is possible that the limit on the left hand side exists, even when the limits on the right hand side fail to exist, or when the right hand side is $\frac{0}{0}$. Therefore, in order to apply these theorems about limits, we must know that the right hand side is not undefined, prior to applying such a lemma. In the proof above, the existence of the limits follows from the induction hypothesis, and that the denominator is nonzero follows from the assumption that division by zero does not occur. This is why we must assume that no division by exactly zero occurs in the evaluation of $f(\epsilon)$ with infinitesimal arithmetic, and it is also why the converse of the theorem does not hold.

7.5.1 *Intervals of Infinitesimal Width Make Paradoxes Disappear*

The proposed observe construct allows finite width intervals `observe(D,(a,w))` where w is an expression that returns a number, as well as infinitesimal width intervals, as in `observe(D,(a,w*eps))` where w is some expression that returns a number and eps is the symbolic infinitesimal ϵ. It is possible to allow higher powers of eps to occur directly in the source program, and it is possible to allow eps to

occur in other places than in widths of intervals, but for conceptual simplicity we shall assume it does not, and that `observe` is always of one of those two forms. That is, we will assume that `eps` is only used in order to translate exact conditioning `observe(D,x)` to `observe(D,(x,w*eps))`.

We translate the example from the introduction as follows:

```
h = rand(Normal(170, 50))
if rand(Bernoulli(0.5))
    observe(Normal(200, 10), (h,w*eps))
end
```

Where the pair `(h,w*eps)` represents an interval of width `w*eps` centered around `h`, in order to condition on the observation to be "exactly `h`".

Let us now investigate the meaning of this program according to the rejection sampling interpretation of `observe`. Assuming the coin flip results in $true$, we reject the trial if the sample from $Normal(200, 10)$ does not fall in the interval $[h - \frac{1}{2}w\epsilon, h + \frac{1}{2}w\epsilon]$. If the coin flip results in $false$, we always accept the trial. If we let $\epsilon \to 0$ then the probability of rejecting the trial goes to 1 if the coin flips to $true$, so almost all successful trials will be those where the coin flipped to $false$. Therefore the expected value of h converges to 170 as $\epsilon \to 0$, and expected value of running this program should be 170.

We translate the BMI example as follows:

```
h = rand(Normal(170, 50))
w = rand(Normal(70, 30))
if rand(Bernoulli(0.5))
    observe(Normal(200, 10), (h, A*eps))
else
    observe(Normal(90, 5), (w, B*eps))
end
bmi = w / h^2
```

Where `A` and `B` are constants with units cm and kg, respectively. The units force us to introduce these constants: since `(h, A*eps)` represents an interval centered at h (in cm), the width `A*eps` must also be a quantity in cm. If we change the units of h or w, we also need to change the units of `A` or `B`. If we change the units of h and `A` from centimeters to meters, the numerical value of h and `A` will both get multiplied by $\frac{1}{100}$. This additional factor for `A*eps`, which cannot be provided in the original non-interval type of `observe(D,x)` statement, is what will make this program behave consistently under change of units.

Both branches of the if statement contain observes with intervals of infinitesimal width, so with rejection sampling both branches will be rejected with probability 1,

regardless of the outcome of the coin flip. We must therefore interpret the example with eps tending to 0, but not being exactly 0. If we chose A to be 1 meter, and B to be 1 kg, and change B to be 1000 kg, then the observe in the else branch is 1000x more likely to succeed compared to before, because the width of the interval goes from 1*eps to 1000*eps. If we made this change then most of the successful trials would be trials where the coin flipped to false. Thus even in the infinitesimal case, the *relative* sizes of the intervals matter a great deal. The relative sizes of the intervals are an essential part of the probabilistic program, and omitting them will inevitably lead to unit anomalies, because changing units also requires resizing the intervals by a corresponding amount (by 1000× in case we change w from kg to g). If we do not resize the intervals, that changes the relative rejection rates of the branches, or the relative weights of the trials, and thus the estimated expectation value $\mathbb{E}[\mathrm{bmi}]$. As Jaynes notes, conditioning on measure-zero events is ambiguous; even though in the limit the intervals (w,1*eps) and (w,1000*eps) both tend to the singleton set {w}, relative to the interval (h,A*eps) it matters *which* of these limits is intended, and the final result will and must depend on which limit was specified.

We translate the third example as follows:

```
x = rand(Normal(10,5))
observe(Normal(15,5), (x,eps))
return exp(x)
```

After a parameter transformation from x to exp(x) we get the following program:

```
exp_x = rand(LogNormal(10,5))
observe(LogNormal(15,5), (exp_x,exp_x*eps))
return exp_x
```

Note that the width of the interval is now exp_x*eps and not simply eps. In general, if we apply a differentiable function f to an interval of width ϵ around x, we obtain an interval of width $f'(x)\epsilon$ around f(x). If we take the exponential of an interval of small width ϵ around x, we get an interval of width $\exp(x)\epsilon$ around exp(x), **not** an interval of width ϵ around exp(x). Both of these programs should give the same estimate for the expectation value of exp(x), so that infinitesimal width intervals allow us to correctly express non-linear parameter transformations without running into Borel-Komolgorov-type paradoxes.

7.5.2 *On the Meaning of "Soft Conditioning"*

It is debatable whether conditioning on small but finite width intervals is preferable to conditioning on measure zero events. Real measurement devices do not measure values to infinite precision. If a measurement device displays 45.88, we might take that to mean an observation in the interval [45.875, 45.885]. The measurement

may in addition measure the true value x plus some `Normal(0,sigma)` distributed noise rather than the true value x. In this case it might be appropriate to use `observe(Normal(x,sigma), (45.88, 0.01))`. The finite precision of the device and its noisy measurement are in principle two independent causes of uncertainty. The rejection sampling interpretation of this program is that we first sample a value from `Normal(x,sigma)` and then continue with the current trial if this lies in the interval $[45.875, 45.885]$, which matches the two sources of uncertainty. An argument for using infinitesimal width intervals is that `observe` on a finite interval requires the evaluation of the distribution's CDF, which is usually more complicated and expensive to compute than the distribution's PDF.

The term "soft conditioning" is sometimes used for `observe(D,x)` statements, particularly when the distribution D is the normal distribution. This term can be interpreted as an alternative to the rejection sampling interpretation in several ways:

1. Rather than conditioning on x being exactly y, we instead condition on x being "roughly" y.

2. The statement `observe(D,x)` means that we continue with the current trial with probability `pdf(D,x)` and reject it otherwise.

We argue that neither of these interpretations is completely satisfactory. For (1) it is unclear what the precise probabilistic meaning of conditioning on x being "roughly" y is. One possible precise meaning of that statement is that we reject the trial if the difference $|x - y|$ is too large, and continue otherwise, but this is not what a statement such as `observe(Normal(y,0.01), x)` does. Rather, it weighs trials where x is close to y higher, and smoothly decreases the weight as the distance between x and y gets larger. It may seem that (2) makes this idea precise, but unfortunately `pdf(D,x)` is not a probability but a probability density, and can even have units or be larger than 1. Furthermore, the statement "continue with the current trial with probability `pdf(D,x)`" seems to have nothing to do with the distribution D as a probability distribution, and instead seems to be a statement that suggests that the statistical model is a biased coin flip rather than drawing a sample from D. Indeed, under our rejection sampling interpretation, if one wants to have a program whose statistical model is about coin flips, one can use the program `observe(Bernoulli(f(x)), true)`. That program *does* mean "flip a biased coin with heads probability `f(x)` and continue with the current trial if the coin landed heads". This makes sense for any function `f(x)` provided the function gives us a probability in the range $[0, 1]$. If that function has a roughly bump-like shape around y, then this will indeed in some sense condition on x being roughly y. The function $C \exp((x - A)^2/B)$ similar to the PDF of the normal distribution does have a bump-like shape around A, so it is possible to use that function for `f`, if one makes sure that B and C are such that it is unitless and everywhere less than 1 (note that this normalization is not the same as the normalization that makes its integral sum to 1).

We therefore suggest to stick with the rejection sampling interpretation of observe statements, and suggest that a statistician who wants to do "soft

conditioning" in the senses (1) and (2) writes their probabilistic program using `observe(Bernoulli(f(x)), true)` where `f` is a function of the desired soft shape rather than `observe(D,x)` where the PDF of `D` has that shape.

7.5.3 *Importance Sampling with Infinitesimal Probabilities*

To do importance sampling for programs with infinitesimal width intervals we need to change almost nothing. We execute a call `observe(D,I)` as `weight *= P(D,I)` where `P(D,I)` has been defined in (2). Since `P(D,I)` returns an infinitesimal number if the width of `I` is infinitesimal, the computed `weight` variable will now contain a symbolic infinitesimal number $r\epsilon^n$ (where n is allowed to be 0), rather than a real number. It will accumulate the product of some number of ordinary probabilities (for `observe` on discrete distributions or continuous distributions with an interval of finite width) and a number of infinitesimal probabilities (for `observe` on continuous distributions with intervals of infinitesimal width).

We now simply evaluate the estimate for $\mathbb{E}[V]$ using the usual weighted average formula, with infinitesimal arithmetic

$$\mathbb{E}[V] \approx \frac{\sum_{k=0}^{N}(weight_k)\cdot(V_k)}{\sum_{k=0}^{N}(weight_k)} \tag{3}$$

In the denominator we are adding numbers of the form $weight_k = w_k\epsilon^{n_k}$. Only the numbers with the minimum value $n_k = n_{min}$ matter; the others are infinitesimally small compared to those, and do not get taken into account due to the definition of $(+)$ on infinitesimal numbers. The same holds for the numerator: the values V_k associated with weights that are infinitesimally smaller do not get taken into account (an optimized implementation could reject a trial as soon as `weight` becomes infinitesimally smaller than the current sum of accumulated weights, since those trials will never contribute to the estimate of $\mathbb{E}[V]$). Therefore the form of the fraction is

$$\mathbb{E}[V] \approx \frac{A\epsilon^{n_{min}}}{B\epsilon^{n_{min}}} = \frac{A}{B}\epsilon^{n_{min}-n_{min}} = \frac{A}{B}\epsilon^0$$

that is, the infinitesimal factors cancel out in the estimate for $\mathbb{E}[V]$, and we obtain a non-infinitesimal result.

We shall now suppose that the symbolic infinitesimal `eps` only occurs in the width of intervals in `observe(D,(x,r*eps))` calls, and not, for instance, in the return value of the probabilistic program. In this case, the estimate (3) of $\mathbb{E}[V]$ satisfies the conditions of Theorem 7.5.5. The calculated estimate may be viewed as a probability expression $f(\epsilon)$ of ϵ (Definition 7.5.4), and since $f(\epsilon) = \frac{A}{B}\epsilon^0$, the theorem implies that $\lim_{x\to0}f(x) = \frac{A}{B}$. Therefore the estimate calculated by importance sampling with infinitesimal arithmetic indeed agrees with taking the limit $\epsilon \to 0$. Figure 44 shows three example probabilistic programs that are parameterized by the interval width.

The blue lines show several runs of the probabilistic program as a function of the interval width, and the orange line shows the result when taking the width to be ϵ. Taking the width to be exactly 0 results in division by zero in the weighted average, but taking it to be ϵ correctly computes the limit: the blue lines converge to the orange lines as the width goes to 0.

7.5.4 *Observe on Points and on Intervals*

We may take a program written using `observe(D,x)` with exact conditioning on points, and convert it to our language by replacing such calls with `observe(D,(x,w*eps))` where `w` is some constant to make the units correct. For programs that exhibit a paradox of type 1 by executing a different number of observes depending on the outcome of calls to `rand`, the computed expectation values will change. However, for programs that always execute the same number of `observe` calls, regardless of the outcome of `rand` calls, the computed expectation values will not be affected by this translation. To see this, note that a call to `observe(D,x)` will multiply `weight *= pdf(D,x)`, whereas `observe(D,(x,w*eps))` will multiply `weight *= pdf(D,x)*w*eps`. Thus if the observe calls are the same in all trials, the only difference is that weight will contain an extra factor of $w\epsilon$ in all trials. The net result is that both the numerator and denominator in the weighted average get multiplied by the factor $w\epsilon$, which has no effect. Thus this translation is conservative with respect to the old semantics, in the sense that it does not change the result that already well-behaved probabilistic programs compute.

7.5.5 *Parameter Transformations as a Language Feature*

The three paradoxes we identified all have to do with parameter transformations. We explicitly add parameter transformations as a language feature. A parameter transformation T allows us to transform a probability distribution D to $T(D)$, such that sampling from $T(D)$ is the same as sampling from D and then applying the function T to the result. In order to ensure that the distribution $T(D)$ has a probability density function we require T to be continuously differentiable. We can also use a parameter transformation to transform an interval from I to $T(I) = \{T(x) : x \in I\}$ which contains all the numbers $T(x)$ for $x \in I$. In order to ensure that the transformed interval is again an interval, we require that T is monotone, that is, whenever $a < b$ we also have $T(a) < T(b)$. In this case, T's action on an interval $[a, b]$ is simple: $T([a, b]) = [T(a), T(a)]$.

Definition 7.5.6. A parameter transformation $T : \mathbb{R}_{\mathbb{A}} \to \mathbb{R}_{\mathbb{B}}$ is a continuously differentiable function with $T'(x) > 0$ for all $x \in \mathbb{R}_{\mathbb{A}}$, where $\mathbb{R}_{\mathbb{A}} \subseteq \mathbb{R}$ and $\mathbb{R}_{\mathbb{B}} \subseteq \mathbb{R}$ are intervals representing its domain and range.

```
function example1(w)
 height=rand(Normal(1.70,0.2))
 weight=rand(Normal(70,30))
 if rand(Bernoulli(0.5))
   observe(Normal(2.0,0.1),
     Interval(height,10*w))
 else
   observe(Normal(90,5),
     Interval(weight,w))
 end
 return weight / height^2
end
```
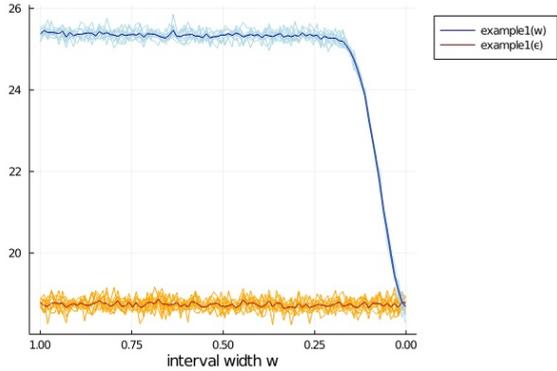
```
function example2(w)
 height=rand(Normal(1.7,0.5))
 if rand(Bernoulli(0.5))
   observe(Normal(2.0,0.1),
     Interval(height,w))
 end
 return height
end
```
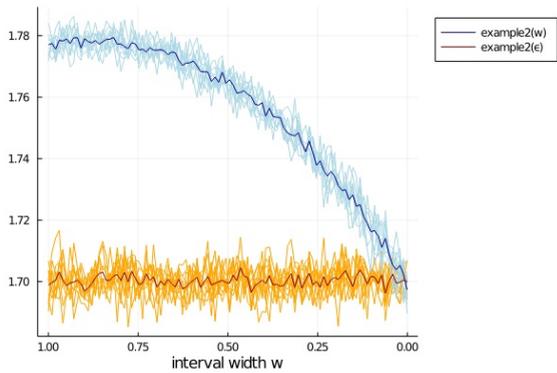
```
function example3(w)
 x=rand(Normal(10,5))
 observe(Normal(15,5),
   Interval(x,w))
 return x
end
```
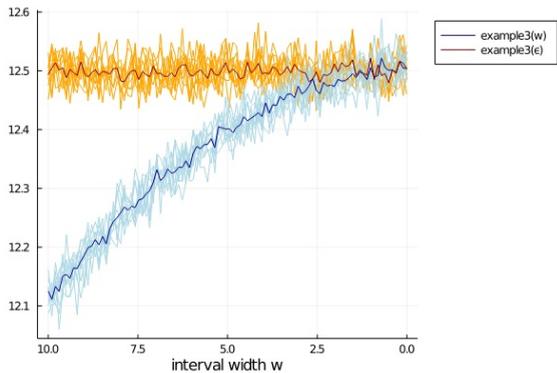

Figure 44: Three example programs evaluated with finite width intervals with width going to zero (blue curves that bend) and with infinitesimal width (orange curves that are horizontal plus noise). The finite width result correctly converges to the infinitesimal result in the limit $w \to 0$. Note that for infinitessimal width, the result no longer depends on w. The graphs show this as a straight line, for comparison with the curve for finite width. The variance is due to choosing a different random seed for each execution.

A strictly monotone function has an inverse on its range, so parameter transformations have an inverse T^{-1} and $T^{-1}(y) = T'(T^{-1}(y))^{-1} > 0$, so the inverse of a parameter transformation is again a parameter transformation.

Example 7.5.7. *The function $T_1(x) = \exp(x)$ is a parameter transformation $T_1 : (-\infty, \infty) \to [0, \infty)$. The function $T_2(x) = 100x$ is a parameter transformation $T_2 : (-\infty, \infty) \to (-\infty, \infty)$.*

The transformation T_1 can be used to convert decibels to energy density, and T_2 can be used to convert meters to centimeters.

Probability distributions need to support 3 operations: random sampling with `rand(D)`, computing the CDF with `cdf(D,x)` and computing the PDF with `pdf(D,x)`. We define these operations for the transformed distribution $T(D)$.

Definition 7.5.8. Given a continuous probability distribution D and a parameter transformation T, we define the operations:

$$\text{rand}(T(D)) = T(\text{rand}(D))$$
$$\text{cdf}(T(D), x) = \text{cdf}(D, T^{-1}(x))$$
$$\text{pdf}(T(D), x) = \text{pdf}(D, T^{-1}(x)) \cdot (T^{-1})'(x)$$

This definition matches how probability distributions transform in probability theory. Our implementation represents a parameter transformation T as the 4-tuple of functions $(T, T', T^{-1}, (T^{-1})')$, so that we have access to the inverse and derivative.

Definition 7.5.9. Given an interval (a, w) with midpoint $a \in \mathbb{R}$ and width $w \in \mathbb{R}$, we let $l = T(a - \frac{w}{2})$ and $r = T(a + \frac{w}{2})$ and define:

$$T((a, w)) = \left(\frac{l + r}{2}, r - l \right)$$

This performs parameter transformation on an interval represented as a midpoint-width pair. If the width is infinitesimal, we need a different rule.

Definition 7.5.10. Given an interval (a, w) with midpoint $a \in \mathbb{R}$ and infinitesimal width w, we define :

$$T((a, w)) = (T(a), T'(a) \cdot w)$$

This performs parameter transformation on an infinitesimal width interval, which gets transformed to an interval whose width is larger by a factor $T'(a)$. The key lemma about parameter transformations is that they do not affect the value of the (possibly infinitesimal) probability of a (possibly infinitesimal) interval.

Lemma 7.5.11. *Let T be a parameter transformation, D a distribution, and I an interval. Then $P(T(D), T(I)) = P(D, I)$ where P is the probability function defined at (2).*

Proof. We distinguish non-infinitesimal intervals from infinitesimal intervals.

- If $I = (a, w)$ is non infinitesimal, then by Definition (2):

$$P(D, (a, w)) = \text{cdf}(D, a + \frac{1}{2}w) - \text{cdf}(D, a - \frac{1}{2}w)$$

  For $T((a, w))$ we have, where $l = T(a - \frac{w}{2})$ and $r = T(a + \frac{w}{2})$:

$$T((a, w)) = (\frac{l + r}{2}, r - l)$$

  and by (2):

$$
\begin{aligned}
P(T(D), T((a, w))) &= \text{cdf}(T(D), \frac{l + r}{2} + \frac{1}{2}(r - l)) - \text{cdf}(T(D), \frac{l + r}{2} - \frac{1}{2}(r - l)) \\
&= \text{cdf}(T(D), r) - \text{cdf}(T(D), l) \\
&= \text{cdf}(D, T^{-1}(r)) - \text{cdf}(D, T^{-1}(l)) \\
&= \text{cdf}(D, T^{-1}(T(a + \frac{w}{2}))) - \text{cdf}(D, T^{-1}(T(a - \frac{w}{2}))) \\
&= \text{cdf}(D, a + \frac{w}{2}) - \text{cdf}(D, a - \frac{w}{2})
\end{aligned}
$$

- If $I = (a, r\epsilon^n)$ is infinitesimal ($n > 0$), then by definition (2):

$$P(D, (a, w)) = \text{pdf}(D, x) \cdot r\epsilon^n$$

  For $T((a, r\epsilon^n))$ we have:

$$T((a, r\epsilon^n)) = (T(a), T'(a) \cdot r\epsilon^n)$$

  and by (2):

$$
\begin{aligned}
P(T(D), T((a, r\epsilon^n))) &= \text{pdf}(T(D), T(a)) \cdot T'(a) \cdot r\epsilon^n \\
&= \text{pdf}(D, T^{-1}(T(a))) \cdot (T^{-1})'(T(a)) \cdot T'(a) \cdot r\epsilon^n \\
&= \text{pdf}(D, a) \cdot r\epsilon^n
\end{aligned}
$$

$\square$

This lemma implies that the effect of `observe(T(D),T(I))` is the same as `observe(D,I)`, since `observe(D,I)` does `weight *= P(D,I)`. This property of `observe` ensures the absence of parameter transformation paradoxes, not only of the three examples we gave, but in general: it does not matter which parameter scale we use; the weight accumulated remains the same.

We have implemented the constructs described in the preceding sections as a simple embedded DSL in the Julia programming language, with the following interface:

- Infinitesimal numbers $r\epsilon^n$ constructed by `Infinitesimal(r,n)`, with predefined `eps = Infinitesimal(1.0,1)`, and overloaded infinitesimal arithmetic operations `+,-,*,/` according to Definition 7.5.2.

- Probability distributions `D` with random sampling `rand(D)` and `cdf(D,x)` and `pdf(D,x)`. These distributions are provided by Julia's Distributions package, which supports beta, normal, Cauchy, Chi-square, Bernoulli, Binomial, and many other continuous distributions and discrete distributions.

- Intervals constructed by `Interval(mid,width)`, where `width` may be infinitesimal, and an operation `P(D,I)` to compute the (possibly infinitesimal) probability that a sample from D lies in the interval I. If I is infinitesimal, then this uses the PDF, and if I has finite width, then this uses the CDF, according to Definition 7.5.3.

- Parameter transformations `T` represented as 4-tuples $(T, T', T^{-1}, (T^{-1})')$, with operations `T(D)` and `T(I)` to transform probability distributions and intervals, according to Definitions 7.5.8, 7.5.9, and 7.5.10.

- The main operations of probabilistic programming DSL are the following:
  - `rand(D)`, where `D` is a distribution provided by Julia's Distributions package.
  - `observe(D,I)`, where `D` is a continuous distribution and `I` is an interval, or `D` is a discrete distribution and `I` is an element, implemented as `weight *= P(D,I)`
  - `importance(trials,program)` which does importance sampling, where `trials` is the number of trials to run, and `program` is a probabilistic program written as a Julia function that uses `rand` and `observe`, and returns the value that we wish to estimate the expectation value of. Importance sampling is implemented as described in Section 7.5.3.

The example in the introduction can be written as follows:

```
function example1_m()
    h = rand(Normal(1.7,0.5))
    if rand(Bernoulli(0.5))
        observe(Normal(2.0,0.1), Interval(h,eps))
    end
    return h
end
estimate = importance(1000000,example1_m)
```

This program will produce an estimate very close to 1.7. If we change the units to centimeters, we will get an estimate very close to 170, as expected:

```
function example1_cm()
    h = rand(Normal(170,50))
    if rand(Bernoulli(0.5))
        observe(Normal(200,10), Interval(h,100*eps))
    end
    return h
end
estimate = importance(1000000,example1_cm)
```

The artifact contains the other examples from the paper and further examples to illustrate the use of the DSL (Jacobs, 2020).

7.7 CONCLUSION & FUTURE WORK

We have seen that naive likelihood accumulation results in unit anomalies when observe statements with continuous distributions are executed conditionally on random data, and we have shown that the culprit is the use of probability densities. From an analysis of what observe statements mean in the discrete case, we motivated a switch to interval-based observe statements, which have a probabilistic and rejection sampling interpretation. To recover the behavior of measure-zero observe statements we introduced intervals with infinitesimal width. This results in the accumulation of infinitesimal probabilities rather than probability densities, which solves the unit anomalies and paradoxes even when conditioning on events of measure zero. Infinitesimal probabilities also enabled us to implement parameter transformations that do not change the behavior of the program. We implemented this form of probabilistic programming as an embedded DSL in Julia.

This improves the state of the art in two ways:

1. It fixes unit and parameter transformation paradoxes, which result in surprising and in some cases arguably incorrect behavior in existing probabilistic programming languages when continuous observe statements are executed conditionally on random data, or when nonlinear parameter transformations are performed.

2. It gives the observe statement a probabilistic and rejection sampling interpretation, with measure zero conditioning as a limiting case when the observation interval is of infinitesimal width.

We hope that this will have a positive impact on the development of the formal semantic foundations of probabilistic programming languages, potentially reducing the problem of conditioning to events of positive measure. On the implementation side, we hope to generalize more powerful inference algorithms such as Metropolis-Hastings and SMC to work with infinitesimal probabilities.

Part IV

GENERAL AND EFFICIENT AUTOMATA
MINIMIZATION

Chapter 8

Fast Coalgebraic Bisimilarity Minimization

ABSTRACT   Coalgebraic bisimilarity minimization generalizes classical automaton minimization to a large class of automata whose transition structure is specified by a functor, subsuming strong, weighted, and probabilistic bisimilarity. This offers the enticing possibility of turning bisimilarity minimization into an off-the-shelf technology, without having to develop a new algorithm for each new type of automaton. Unfortunately, there is no existing algorithm that is fully general, efficient, and able to handle large systems.

We present a generic algorithm that minimizes coalgebras over an arbitrary functor in the category of sets as long as the action on morphisms is sufficiently computable. The algorithm makes at most $O(m \log n)$ calls to the functor-specific action, where n is the number of states and m is the number of transitions in the coalgebra.

While more specialized algorithms can be asymptotically faster than our algorithm (usually by a factor of $O(\frac{m}{n})$), our algorithm is especially well suited to efficient implementation, and our tool *Boa* often uses much less time and memory on existing benchmarks, and can handle larger automata, despite being more generic.

8.1   INTRODUCTION

State-based systems arise in various shapes throughout computer science: as automata for regular expressions, as control-flow graphs of programs, Markov decision processes, (labelled) transition systems, or as the small-step semantics of programming languages. If the programming language of interest involves concurrency, bisimulation can capture whether two systems exhibit the same behavior (Winskel, 1993; Milner, 1980). In model checking, a state-based system is derived from the implementation and then checked against its specification.

It is often beneficial to reduce the size of a state-based system by merging all equivalent states. Moore's algorithm (Moore, 1956) and Hopcroft's $O(|\Sigma| \, n \log n)$ algorithm (Hopcroft, 1971) do this for the deterministic finite automata that arise from regular expressions over alphabet Σ, and produce the equivalent automaton with minimal number of states. In model checking, state-space reduction can be effective as a preprocessing step (Baier and Katoen, 2008). For instance, in probabilistic model checking, the time saved in model checking due to the smaller system exceeds the time needed to minimize the system (Katoen et al., 2007).

Subsequent to Hopcroft (1971), a variety of algorithms were developed for minimizing different types of automata. Examples are algorithms for

- transition systems (without action labels) (Kanellakis and Smolka, 1983, 1990), labelled transition systems (Valmari, 2009), which arise from the verification of concurrent systems,

- weighted bisimilarity (Valmari and Franceschinis, 2010) for Markov chains and probabilistic settings (such as probabilistic model checking (Katoen et al., 2007)),

- Markov decision processes (Baier et al., 2000; Groote et al., 2018) that combine concurrency with probabilistic branching,

- weighted tree automata (Björklund et al., 2009, 2007) that arise in natural language processing (May and Knight, 2006).

Recently, those algorithms and system equivalences were subsumed by a coalgebraic generalization (Dorsch et al., 2017; Deifel et al., 2019; Wißmann et al., 2021). This generic algorithm is parametrized by a (Set-)functor that describes the concrete system type of interest. Functors are a standard notion in category theory and a key notion in the Haskell programming language. In coalgebraic automaton minimization, the functor is used to attach transition data to each state of the automaton. For instance, the powerset functor models non-deterministic branching in transition systems, and the probability distribution functor models probabilistic branching in Markov chains.

The users of a coalgebraic minimization algorithm may create their own system type by composing the provided basic functors, allowing them to freely combine deterministic, non-deterministic, and probabilistic behavior. For instance, the functor to model Markov decision processes is the composition of the functors of transition systems and the functor for probability distributions. This generalization points to the enticing possibility of turning automata minimization for different types of automata into an off-the-shelf technology.

Unfortunately, there are two problems that currently block this vision. Firstly, although the generic algorithm has excellent $O(m \log n)$ asymptotic complexity, where n is the number of states and m is the number of edges, it is slow in practice, and the data structures required for partition refinement suffer from hungry memory usage. A machine with 16GB of RAM required several minutes to minimize tree automata with 150 thousand states and ran out of memory when minimizing tree automata larger than 160 thousand states (Deifel et al., 2019; Wißmann et al., 2021). This problem has also been observed for algorithms for specific automata types, e.g., transition systems (Valmari, 2010). In order to increase the total memory available, a distributed partition refinement algorithm has been developed (Birkmann et al., 2022), (and previously also for specific automata types, e.g., labelled transition systems (Blom and Orzan, 2005)), but this algorithm runs in $O(n^2)$ and requires expensive distributed hardware.

Secondly, the generic algorithm does not work for all Set-functors, because it places certain restrictions on the functor type necessary for the tight run time complexity. For instance, the algorithm is not capable of minimizing frames for the monotone

neighbourhood logic (Hansen and Kupke, 2004), arising in game theory (Parikh, 1985; Peleg, 1987; Pauly, 2001).

We present a new algorithm that works for *all* system types given by computable Set-functors, requiring only an implementation of the functor's action on morphisms, which is then used to compute so-called *signatures of states*, a notion originally introduced for labelled transition systems (Blom and Orzan, 2005). The algorithm makes at most $O(m \log n)$ calls to the functor implementation, where n and m are the number of states and edges in the automaton, respectively. In almost all instances, one such call takes $O(k)$ time, where k is the maximum out-degree of a state, so the overall run time is in $O(km \log n)$. We compensate for this extra factor because our algorithm has been designed to be efficient in practice and does not need large data structures: we only need the automaton with predecessors and a refinable partition data structure.

We provide an implementation of our algorithm in our tool called *Boa*. The user of the tool can either encode their system type as a composition of the functors natively supported by *Boa*, or extend *Boa* with a new functor by providing a small amount of Rust code that implements the functor's action on morphisms.

Empirical evaluation of our implementation shows that the memory usage is much reduced, in certain cases by more than 100x compared to the distributed algorithm (Birkmann et al., 2022), such that the benchmarks that were used to illustrate its scalability can now be solved on a single computer. Running time is also much reduced, in certain cases by more than 3000x, even though we run on a single core rather than a distributed cluster. We believe that this is a major step towards coalgebraic partition refinement as an off-the-shelf technology for automaton minimization.

THE REST OF THE PAPER IS STRUCTURED AS FOLLOWS.

- Section 8.2: Coalgebraic bisimilarity minimization and our algorithm in a nutshell.

- Section 8.3: The formal statement of behavioral equivalence of states, and examples for how this reduces to known notions of equivalence for particular instantiations.

- Section 8.4: Detailed description of our coalgebraic minimization algorithm for any computable set functor, and time complexity analysis showing that the algorithm makes at most $O(m \log n)$ calls to the functor operation.

- Section 8.5: Instantiations of the algorithm showing its genericity.

- Section 8.6: Benchmark results showing our algorithm outperforms earlier work.

- Section 8.7: Conclusion and future work.

8.2 FAST COALGEBRAIC BISIMILARITY MINIMIZATION IN A NUTSHELL

This section presents the key ideas of our fast coalgebraic minimization algorithm. We start with an introduction to coalgebra, and how the language of category theory provides an elegant unifying framework for different types of automata. No knowledge of category theory is assumed; we will go from the concrete to the abstract, and category theoretic notions have been erased from the presentation as much as possible.

Let us thus start by looking at three examples of automata: deterministic finite automata on the alphabet $\{a, b\}$, transition systems, and Markov chains. The usual way of visualizing is depicted in the first row of Figure 45. For instance, a deterministic finite automaton on state set C is usually described via a transition function $\delta\colon C \times \{a, b\} \to C$ and a set of accepting states $F \subseteq C$ (the initial state is not relevant for the task of computing equivalent states). In order to generalize various types of automata, however, we take a *state-centric* point of view, where we consider all the data as being *attached to a particular state*:

- In a finite automaton on the alphabet $\{a, b\}$ each state has two successors: one for the input letter a and one for the input letter b. Each state also carries a boolean that determines whether the state is accepting (double border), or not (single border). For instance, state **3** in the deterministic automaton in the left column of Figure 45 is not accepting, but after transitioning via a it goes to state **5**, which is accepting. We can specify any deterministic automaton entirely via a map

$$c\colon C \to \{F, T\} \times C \times C$$

  This map sends every state $q \in C$ to $(x, q_a, q_b) := c(q)$, where $x \in \{F, T\}$ specifies if q is accepting, and $q_a, q_b \in C$ are the target states for in input a and b, respectively.

- A transition system consists of a (finite) set of locations C, plus a (finite) set of transitions "$\to$" $\subseteq C \times C$. For instance, state **3** in the figure can transition to state **4** or **5** or to itself, whereas **5** cannot transition anywhere. A transition system is specified by a map
$$c\colon C \to \mathcal{P}_f(C)$$
  where $\mathcal{P}_f(C)$ is the set of finite subsets of C. This map sends every location q to the set of locations $c(q) \subseteq C$ to which a transition exists.

- A Markov chain consists of a set of states, and for each state a probability distribution over all states describes the transition behavior. That is, for each pair of states $q, q' \in C$, the probability $p_{q,q'} \in [0, 1]$ denoting the probability to transition from q to q'. We also attach a boolean label to each state (again, indicated by double border). For instance, state **1** in the figure steps to state **2**

	DFA	Transition system	Markov chain
Functor	$F(X) = \{F, T\} \times X \times X$	$F(X) = \mathcal{P}_f(X)$	$F(X) = \{F, T\} \times \mathcal{D}(X)$
Coalgebra $c \colon C \to F(C)$	$1 \mapsto (F, 2, 3)$ $2 \mapsto (F, 4, 3)$ $3 \mapsto (F, 5, 3)$ $4 \mapsto (T, 5, 4)$ $5 \mapsto (T, 4, 4)$	$1 \mapsto \{2, 3, 4\}$ $2 \mapsto \{1, 4\}$ $3 \mapsto \{3, 4, 5\}$ $4 \mapsto \{4, 5\}$ $5 \mapsto \{\ \}$	$1 \mapsto (F, \{2 : \frac{1}{3}, 3 : \frac{2}{3}\})$ $2 \mapsto (F, \{2 : \frac{1}{2}, 4 : \frac{1}{2}\})$ $3 \mapsto (F, \{2 : \frac{1}{4}, 4 : \frac{1}{2}, 5 : \frac{1}{4}\})$ $4 \mapsto (T, \{4 : 1\})$ $5 \mapsto (F, \{3 : \frac{1}{2}, 4 : \frac{1}{2}\})$
Equivalence	$2 \equiv 3, 4 \equiv 5$	$1 \equiv 2, 3 \equiv 4$	$2 \equiv 3 \equiv 5$
Minimized $c' \colon C' \to F(C')$	$1 \mapsto (F, 2, 2)$ $2 \mapsto (F, 4, 2)$ $4 \mapsto (T, 4, 4)$	$1 \mapsto \{1, 3\}$ $3 \mapsto \{3, 5\}$ $5 \mapsto \{\ \}$	$1 \mapsto (F, \{2 : 1\})$ $2 \mapsto (F, \{2 : \frac{1}{2}, 4 : \frac{1}{2}\})$ $4 \mapsto (T, \{4 : 1\})$

Figure 45: Examples of different system types and their encoding as coalgebras for the state set $C = \{1, 2, 3, 4, 5\}$.

with probability $\frac{1}{3}$ and to state 3 with probability $\frac{2}{3}$. Such a Markov chain is specified by a map

$$c \colon C \to \{F, T\} \times \mathcal{D}(C)$$

where $\mathcal{D}(C)$ is the set of finite probability distributions over C.

We call the data $c(q)$ attached to a state q the **successor structure** of the state q.

By generalizing the pattern above, different types of automata can be treated in a uniform way: In all these examples, we have a set of states C (where $C = \{1, 2, 3, 4, 5\}$ in the figure), and then a map $c \colon C \to F(C)$ for the successor structures, for some construction F turning the set of states C into another set $F(C)$. Such a mapping $F \colon \mathsf{Set} \to \mathsf{Set}$ (in programming terms one should think of F as a type constructor) is called a functor, and describes the automaton type. This point of view allows us to easily consider variations, such as labelled transition systems, given by $F(X) = \mathcal{P}_f(\{a, b\} \times X)$, and Markov chains where the states are not labelled but the transitions are labelled, given by $F(X) = \mathcal{D}(\{a, b\} \times X)$. Other examples, such as monoid weighted systems, Markov Decision processes, and tree automata, are given in Section 8.3. Representing an automaton of type F by attaching a successor structure of type $F(C)$ to each state $q \in C$ brings us to the following definition:

Definition 8.2.1. An automaton of type F, or finite F-coalgebra, is a pair (C, c) of a finite set of states C, and a function $c\colon C \to F(C)$ that attaches the successor structure of type $F(C)$ to each state in C.

Since C is a finite set of states, we can give such a map c by listing what each state in C maps to. For the concrete automata in Figure 45, the representation using such a mapping $c\colon C \to F(C)$ is given in the "Coalgebra" row.

8.2.1 *Behavioral Equivalence of States in F-automata, Generically*

We now know how to uniformly *represent* an automaton of type F, but **we need a uniform way to state what it means for states to be equivalent**. Intuitively, we would like to say that two states are equivalent if the successor structures attached to the two states by the map $c\colon C \to F(C)$ are equivalent. The difficulty is that the successor structure may itself contain other states, so equivalence of states requires equivalence of successor structures and vice versa.

A way to cut this knot is to consider a *proposed* equivalence of states, and then define what it means for this equivalence to be valid, namely: an equivalence of states is *valid* if proposed to be equivalent states have equivalent successor structures, where equivalence of the successor structures is considered up to the *proposed* equivalence of states. In short, the proposed equivalence should be compatible with the transition structure specified by the successor structures.

Rather than representing a proposed equivalence as an equivalence relation $R \subseteq C \times C$ on the state space C, it is better to use a surjective map $r\colon C \to C'$ that assigns to each state a canonical representative in C' identifying its equivalence class (also called *block*). That is, two states q, q' are equivalent according to r, if $r(q) = r(q')$. Intuitively, r partitions the states into blocks or equivalence classes $\{q \in C \mid r(q) = y\} \subseteq C$ for each canonical representative $y \in C'$. Not only does this representation of the equivalence avoid quadratic overhead in the implementation, but it is also more suitable to state the stability condition:

An equivalence $r\colon C \to C'$ is *stable*, if for every two equivalent states q_1, q_2 (i.e., with $r(q_1) = r(q_2)$), the successor structures $c(q_1)$ and $c(q_2)$ attached to the states become *equal* after replacing states q inside the successor structures with their canonical representative $r(q)$.

This guarantees that we can build a minimized automaton with the canonical representatives $r(q) \in C'$ as state space. If we do this replacement for both the source and the target of all transitions, we obtain a potentially smaller automaton $c'\colon C' \to F(C')$.

In order to gain intuition about this, let us investigate our three examples in Figure 45:

- In the finite automaton, the states $4 \equiv 5$ and $2 \equiv 3$ can be shown to be equivalent, so we have $C' = \{1, 2, 4\}$ and $r\colon C \to C'$ with $3 \mapsto 2$ and $5 \mapsto 4$ (and also $1 \mapsto 1$, $2 \mapsto 2$, $4 \mapsto 4$, which we will use implicitly in future examples). We can check that

this equivalence is compatible with c by verifying that the successor structures of supposedly equivalent states become *equal* after substituting $5 \mapsto 4$ and $3 \mapsto 2$. After substituting $5 \mapsto 4$ we indeed have that $c(2) = (F, 4, 3)$ and $c(3) = (F, 5, 3)$ become equal, and that $c(4) = (T, 5, 4)$ and $c(5) = (T, 4, 4)$ become equal. So this equivalence is stable.

- For the transition system, the states $3 \equiv 4$ are equivalent, and $1 \equiv 2$ are equivalent. We can verify, for instance, that states $c(1) = \{2, 3, 4\}$ and $f(2) = \{1, 4\}$ are equivalent, because after substituting $4 \mapsto 3$ and $2 \mapsto 1$, we indeed have $\{1, 3, 3\} = \{1, 3\}$, because duplicates can be removed from sets. Note that it is important that the data for transition systems are sets rather than lists or multisets. Multisets also give a valid type of automaton, but they do not give the same notion of equivalence.

- For the Markov chain, we can verify $2 \equiv 3 \equiv 5$. Consider that all three of these states step to state 4 with probability $\frac{1}{2}$. With the remaining probability $\frac{1}{2}$ these states step to one of the states $2 \equiv 3 \equiv 5$, i.e. they stay in this block. State 3 steps to either state 2 or 5 with probability of $\frac{1}{4}$ each. If we however assume that state 5 behaves equivalent to 2, then the branching of state 3 is the same as going to state 2 with probability $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$ directly. Thus, when substituting $5 \mapsto 2$ and $3 \mapsto 2$ the distribution $c(3) = (F, \{2\colon \frac{1}{4}, 4\colon \frac{1}{2}, 5\colon \frac{1}{4}\})$, collapses to $(F, \{2\colon \frac{1}{2}, 4\colon \frac{1}{2}\})$. In other words, edges to equivalent states get merged by summing up their probability.

Here we assumed that we were given an equivalence, which we check to be stable. Our next task is to determine how to find the maximal stable equivalence. We shall see that this only requires a minor modification to checking that a given equivalence is stable: if we discover that an equivalence is *not* stable, we can use that information to iteratively refine the equivalence until it is stable.

8.2.2 *Minimizing F-automata, Generically: The Naive Algorithm*

In this section we describe a **naive but generic method for minimizing F-automata** (König and Küpper, 2014). The method is based on the observation that we can start by optimistically assuming that *all* states are equivalent, and then use the stability check described in the preceding section to determine how to split up into finer blocks. By iterating this procedure we will arrive at the minimal automaton.

Let us thus see what happens if we blindly assume *all* states to be equivalent, and perform the substitution where we change every state to state 1. For the finite automaton in Figure 45, we get

$$1 \mapsto (F, 1, 1) \qquad 2 \mapsto (F, 1, 1) \qquad 3 \mapsto (F, 1, 1) \qquad 4 \mapsto (T, 1, 1) \qquad 5 \mapsto (T, 1, 1)$$

Clearly, even though we assumed all states to be equivalent, the states $1, 2, 3$ are still distinct from $4, 5$ because the former three are not accepting whereas the latter two

$1 \mapsto (F, 1, 1)$	$1 \mapsto (F, 1, 1)$	$1 \mapsto (F, 2, 2)$
$2 \mapsto (F, 1, 1)$	$2 \mapsto (F, 4, 1)$	$2 \mapsto (F, 4, 2)$
$3 \mapsto (F, 1, 1)$	$3 \mapsto (F, 4, 1)$	$3 \mapsto (F, 4, 2)$
$4 \mapsto (T, 1, 1)$	$4 \mapsto (T, 4, 4)$	$4 \mapsto (T, 4, 4)$
$5 \mapsto (T, 1, 1)$	$5 \mapsto (T, 4, 4)$	$5 \mapsto (T, 4, 4)$

$1 \mapsto \{1\}$	$1 \mapsto \{1\}$	$1 \mapsto \{1, 3\}$
$2 \mapsto \{1\}$	$2 \mapsto \{1\}$	$2 \mapsto \{1, 3\}$
$3 \mapsto \{1\}$	$3 \mapsto \{1, 5\}$	$3 \mapsto \{3, 5\}$
$4 \mapsto \{1\}$	$4 \mapsto \{1, 5\}$	$4 \mapsto \{3, 5\}$
$5 \mapsto \{\,\}$	$5 \mapsto \{\,\}$	$5 \mapsto \{\,\}$

$1 \mapsto (F, \{1 : 1\})$	$1 \mapsto (F, \{1 : 1\})$	$1 \mapsto (F, \{2 : 1\})$
$2 \mapsto (F, \{1 : 1\})$	$2 \mapsto (F, \{1 : \frac{1}{2}, 4 : \frac{1}{2}\})$	$2 \mapsto (F, \{2 : \frac{1}{2}, 4 : \frac{1}{2}\})$
$3 \mapsto (F, \{1 : 1\})$	$3 \mapsto (F, \{1 : \frac{1}{2}, 4 : \frac{1}{2}\})$	$3 \mapsto (F, \{2 : \frac{1}{2}, 4 : \frac{1}{2}\})$
$4 \mapsto (T, \{1 : 1\})$	$4 \mapsto (T, \{4 : 1\})$	$4 \mapsto (T, \{4 : 1\})$
$5 \mapsto (F, \{1 : 1\})$	$5 \mapsto (F, \{1 : \frac{1}{2}, 4 : \frac{1}{2}\})$	$5 \mapsto (F, \{2 : \frac{1}{2}, 4 : \frac{1}{2}\})$

Figure 46: Execution of the naive algorithm for the three automata of Figure 45.

are. Therefore, even if we initially assumed all states to be equivalent, we discover inequivalent states. Let us thus try the equivalence $1 \equiv 2 \equiv 3$ and $4 \equiv 5$, and apply substitution where we send $2 \mapsto 1$, $3 \mapsto 1$ and $5 \mapsto 4$:

$$1 \mapsto (F, 1, 1) \quad 2 \mapsto (F, 4, 1) \quad 3 \mapsto (F, 4, 1) \quad 4 \mapsto (T, 4, 4) \quad 5 \mapsto (T, 4, 4)$$

We have now discovered *three* distinct blocks of states: state 1, states $2 \equiv 3$ and states $4 \equiv 5$. If we apply a substitution for *that* equivalence, we get:

$$1 \mapsto (F, 2, 2) \quad 2 \mapsto (F, 4, 2) \quad 3 \mapsto (F, 4, 2) \quad 4 \mapsto (T, 4, 4) \quad 5 \mapsto (T, 4, 4)$$

We did not discover new blocks; we still have three distinct blocks of states: 1, states $2 \equiv 3$ and states $4 \equiv 5$. Hence, there is no need to change the substitution map sending each state to a representative in the $\equiv$-class, and so we reached a fixed point. We can now read off the minimized automaton by deleting states 3 and 5 from the last automaton above.

The reader may observe that the process sketched above is quite general, and can be used to minimize a large class of automata. The sketch translates into the pseudocode in Algorithm 1.

Algorithm 1 Sketch of the naive partition refinement algorithm

---

procedure NAIVEALGORITHM(automaton)   ▷ *Finds equivalent states of automaton*
    Put all states in one block (*i.e.,* assume that all states are equivalent)
    while number of blocks grows **do**
        Substitute current block numbers in the successor structures
        Split up blocks according to the successor structures

---

The execution trace of this naive algorithm for our three example automata of Figure 45 can be found in Figure 46. What the algorithm only needs is the ability to obtain a canonicalized successor structure after applying a substitution to the successor states. In general this may involve some amount of computation. For instance, for transition systems, a purely textual substitution would lead to $\{1, 1, 1\}$ assuming all states are conjectured equivalent in the first step, and the canonical form of this set is $\{1\}$. Note that the states $1 - 4$ all have successor structure $\{1\}$ in the first step of the algorithm, but they get distinguished from state 5, which has successor structure $\{\ \}$.

We see that in order to talk about equivalence of states, and in order to perform minimization, we need a notion of substitution and canonicalization. As it turns out, this corresponds exactly to the standard definition of functor in category theory (for Set):

Definition 8.2.2. $F \colon \mathsf{Set} \to \mathsf{Set}$ is a functor, if given an $p \colon A \to B$ (i.e., a "substitution"), we have a mapping $F[p] \colon F(A) \to F(B)$. Furthermore, this operation must satisfy $F[\mathrm{id}] = \mathrm{id}$ and $F[p \circ g] = F[p] \circ F[g]$.

We thus require all automata types to be given by functors in the sense of Theorem 8.2.2. We can then talk about equivalence of states, and minimize automata by repeatedly applying this operation $F[p]$ as sketched above. A more formal naive algorithm will be discussed in Section 8.4.2.

8.2.3   *The Challenge: A Generic* and *Efficient Algorithm*

The problem with the naive algorithm sketched in Section 8.2.2 is that it processes all transitions in every iteration of the main loop. In certain cases, partition refinement (in general) may take $\Theta(n)$ iterations to converge, where n is the number of states. This can happen, for instance, if the automaton has a long chain of transitions, so in each iteration, only one state is moved to a different block. Figure 47 contains three example automata for which the naive algorithm takes $\Theta(n)$ iterations (provided one generalizes the examples to have n nodes).

Since naive algorithm computes new successor structures for all states in each iteration, the functor operation is applied $O(n^2)$ times in total. Thus, the challenge we set out to solve is the following:
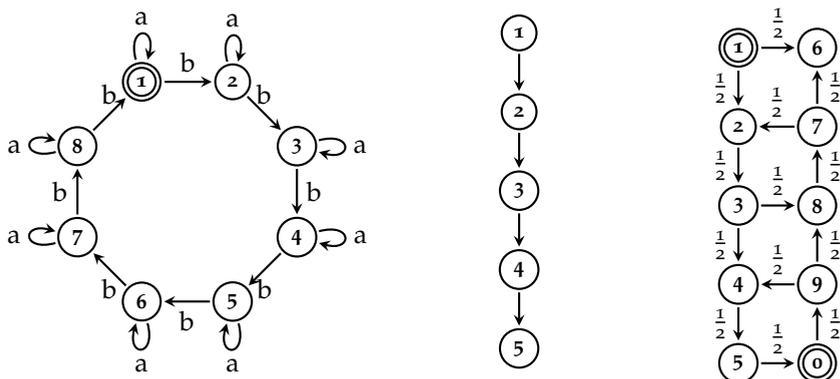
Figure 47: Examples of shapes of automata on which the naive algorithm runs in $\Theta(n^2)$.

Can we find an asymptotically and practically efficient algorithm for automaton minimization that uses only the successor structure recomputation operation $F[p]$?

By using only $F[p]$, we do not impose further conditions on the functor F beside $F[p]$ being computable. Since the algorithm does not inspect F any further, the only condition imposed on the functor is that $F[p]$ is computable for all substitutions p on the state space.

8.2.4 *Hopcroft's Trick: The Key to Efficient Minimization*

A key part of the solution is a principle often called "Hopcroft's trick" or "half the size" trick, which underlies all known asymptotically efficient automata minimization algorithms. To understand the trick, consider the following game:

1. We start with a set of objects, *e.g.,* $\{1,2,3,4,5,6,7,8,9\}$.

2. We chop the set into two parts arbitrarily, *e.g.,* $\{1,3,5,7,9\}, \{2,4,6,8\}$.

3. We select one of the sets, and chop it up arbitrarily again, *e.g.,* $\{1,3\}, \{5,7,9\}, \{2,4,6,8\}$.

4. We continue the game iteratively (possibly until all sets are singletons).

Once the game is complete, we trace back the history of one particular element, say 3, and count how many times it was in the smaller part of a split:

The number of times an element was part of the smaller half of a split is $O(\log n)$.

One can prove this bound by considering the evolution of the size of the set containing the element. Initially, this size is n. Each time the element was part of

the smaller part of the split, the size of the surrounding set gets cut in at least half, which can happen at most $O(\log n)$ times before we reach a singleton.

This indicates that for efficient algorithms, we should make sure that the running time of the algorithm is only proportional to the smaller halves of the splits. In other words, when we split a block, we have to make sure that we do not loop over the larger half of the split.

A slightly more general bound results from considering a game where we can split each set into an arbitrary number of parts, rather than 2:

The number of times an element was part of <u>a</u> smaller part of the split is $O(\log n)$.

In this case, "a smaller part of the split" is to be understood as any part of the split except the largest part. Thus, if we split $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ into $\{1, 3\}, \{5, 7, 9\}, \{2, 4, 6, 8\}$, then $\{1, 3\}, \{5, 7, 9\}$ are both considered "smaller parts", whereas $\{2, 4, 6, 8\}$ is the larger part.

In terms of algorithm design, our goal shall thus be that when we do a k-way split of a block, we may do operations proportional to all the $k - 1$ smaller parts of the split, but never an operation proportional to the largest part of the split.

8.2.5 *A Sketch of our Generic and Efficient Algorithm*

We design our algorithm based on the naive algorithm and Hopcroft's trick. The main problem with the naive algorithm is that it recomputes the successor structures of *all* states at each step. The reader may already have noticed that many of the successor structures in fact stay the same, and are unnecessarily recomputed. The successor structure of a state only changes if the block number of one of its successors changes. *The key to a more efficient algorithm is to minimize the number of times a block number changes, so that successor structure recomputation is avoided as much as possible.*

In the naive algorithm, we see that when we split a block of states into smaller blocks, we have freedom about which numbers to assign to each new sub-block. We therefore choose to *keep the old number for the largest sub-block*. Hopcroft's trick will then ensure that a state's number changes at most $O(\log n)$ times.

In order to reduce recomputation of successor structures, our algorithm tracks for each block of states (*i.e.,* states with the same block number), which of the states are *dirty*, meaning that at least one of their successors' number changed. The remaining states in the block are *clean*, meaning that the successors did not change.

Importantly, all clean states of a block *have the same successor structure*, because (A) their successors did not change (B) if their successor structure was different in the last iteration, they would have been placed in different blocks. Therefore, in order to recompute the successor structures of a block, it suffices to recompute the dirty states and *one* of the clean states, because we know that all the clean states have the same successor structure.

This sketch translates into the pseudocode of Algorithm 2.

Algorithm 2 Sketch of the optimized partition refinement algorithm

---

procedure PARTREFSETFUN(automaton)   ▷ *Finds equivalent states of automaton*
    Put all states in one block (*i.e.,* assume that all states are equivalent)
    Mark all states dirty
    while number of blocks grows **do**
        Pick a block with dirty states
        Compute the successor structures of the dirty states and one clean state
        Mark all states in the block clean
        Split up the block, keeping the old block number for the largest sub-block
        Mark all predecessors of changed states dirty

Let us investigate the complexity of this algorithm in terms of the number of successor structure recomputations. By Hopcroft's trick, a state's number can now change at most $O(\log n)$ times, since we do not change the block number of the largest sub-block. Whenever we change a state's number, all the predecessors of that state will need to be marked dirty, and be recomputed. If we take a more global view, we can see that a recomputation may be triggered for every edge in the automaton, for each time the number of the destination state of the edge changes. Therefore, if there are m edges, there will be at most $O(m \log n)$ successor structure recomputations, *i.e.,* at most $O(m \log n)$ calls to the functor operation.

In order to make the algorithm asymptotically efficient in terms of the total number of primitive computation steps, we must make sure to never do any operation that is proportional to the number of clean states in a block. Importantly, we must be able to split a block into k sub-blocks without iterating over the clean states. To do this, we have to devise efficient data structures to keep track of the blocks and their dirty states (Section 8.4.3).

We implement our algorithm (Section 8.4.4) with these data structures and efficient methods for computing the functor operation in our tool, *Boa*. When using *Boa*, the user can either encode their automata using a composition of the built-in functors, or implement their own functor operation and instantiate the algorithm with that.

Practical efficiency of the algorithm

Previous work on algorithms that apply to classes of functors that support more specialized operations in addition to just the functor operation can give better asymptotic complexity when one considers more fine-grained accounting than just the number of calls to the functor operation (Dorsch et al., 2017; Wißmann et al., 2020; Deifel et al., 2019; Wißmann et al., 2021). Perhaps surprisingly, even though our algorithm is very generic and does not have access to these specialized operations, our algorithm is much faster than the more specialized algorithm in practice (Section 8.6).

However, the limiting factor in practice is not necessarily time but space. The aforementioned algorithm requires on the order of 16GB of RAM for minimizing

automata with 150 thousand states (Deifel et al., 2019; Wißmann et al., 2021). In order to be able to access more memory, distributed algortithms have been developed (Birkmann et al., 2022; Blom and Orzan, 2005). Using a cluster with 265GB of memory, the distributed algorithm was able to minimize an automaton with 1.3 million states and 260 million edges. By contrast, *Boa* is able to minimize the same automaton using only 1.7GB of memory.

The reason is that we do not need any large auxiliary data structures; most of the 1.7GB is used for storing the automaton itself. Furthermore, because we only need to compute the functor operation for states in the automaton, we are able to store the automaton in an efficient immutable binary format.

In the rest of the paper we will first give a more formal definition of bisimilarity in coalgebras (Section 8.3), we describe how we represent our automata, and which basic operations we need (Section 8.4.1), we describe the auxiliary data structures required by our algorithm (Section 8.4.3), we describe our algorithm and provide complexity bounds (Section 8.4.4), we show a variety of functor instances that our algorithm can minimize (Section 8.5), we compare the practical performance to earlier work (Section 8.6), and we conclude the paper (Section 8.7).

8.3 COALGEBRA AND BISIMILARITY, FORMALLY

In this section we define formally what it means for two states in a coalgebra to be behaviorally equivalent, and we give examples to show that behavioral equivalence in coalgebras reduces to known notions of bisimilarity for specific functors.

Recall that we model state-based systems as coalgebras for set functors (Theorem 8.2.2):

Definition 8.3.1. An F-*coalgebra* consists of a carrier set C and a structure map $c \colon C \to FC$.

Intuitively, the carrier C of a coalgebra (C, c) is the set of states of the system, and for each state $x \in C$, the map provides $c(x) \in FC$ that is the structured collection of successor states of x. If $F = \mathcal{P}_f$, then $c(x)$ is simply a finite set of successor states. The functor determines a canonical notion of behavioral equivalence.

Definition 8.3.2. A *homomorphism* between coalgebras $h \colon (C, c) \to (D, d)$ is a map $h \colon C \to D$ with $F[h](c(x)) = d(h(x))$ for all $x \in C$. States x, y in a coalgebra (C, c) are *behaviorally equivalent* if there is some other coalgebra (D, d) and a homomorphism $h \colon (C, c) \to (D, d)$ such that $h(x) = h(y)$.

Example 8.3.3. *We consider coalgebras for the following functors (see also Table 1):*

1. *Coalgebras for $\mathcal{P}_f$ are finitely-branching transition systems and states x, y are behaviorally equivalent iff they are bisimilar.*

2. *An (algebraic)* signature *is a set Σ together with a map* $\mathrm{ar}\colon \Sigma \to \mathbb{N}$. *The elements of* $\sigma \in \Sigma$ *are called* operation symbols *and* $\mathrm{ar}(\sigma)$ *is the arity. Every signature induces a functor defined by*

$$\tilde{\Sigma}X = \{(\sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)}) \mid \sigma \in \Sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)} \in X\}$$

on sets and for maps $f\colon X \to Y$ *defined by*

$$\tilde{\Sigma}[f](\sigma, x_1, \ldots, x_{\mathrm{ar}(\sigma)}) = (\sigma, f(x_1), \ldots, f(x_{\mathrm{ar}(\sigma)})).$$

A state in a $\tilde{\Sigma}$-coalgebra describes a possibly infinite Σ-tree, with nodes labelled by $\sigma \in \Sigma$ with $\mathrm{ar}(\sigma)$ many children. Two states are behaviorally equivalent iff they describe the same Σ-tree.

3. *Deterministic finite automata on alphabet A are coalgebras for the signature Σ with 2 operation symbols of arity $|A|$. States are behaviorally equivalent iff they accept the same language.*

4. *For a commutative monoid $(M, +, 0)$, the* monoid-valued *functor $M^{(X)}$ (*Gumm and Schröder*, 2001, Def. 5.1) can be thought of as M-valued distributions over X:*

$$M^{(X)} := \{\mu\colon X \to M \mid \mu(x) \neq 0 \text{ for only finitely many } x \in X\}$$

The map $f\colon X \to Y$ is sent by $M^{(-)}$ to

$$M^{(f)}\colon M^{(X)} \to M^{(Y)} \qquad M^{(f)}(\mu) = \Big(y \mapsto \sum_{x \in X, f(x)=y} \mu(x)\Big)$$

Coalgebras for $M^{(-)}$ are weighted systems whose weights come from M.

*A coalgebra $c\colon C \to M^{(C)}$, sends a state $x \in C$ and another state $y \in C$ to a weight $m := c(x)(y) \in M$ which is understood as the weight of the transition $x \xrightarrow{m} y$, where $c(x)(y) = 0$ is understood as no transition. The coalgebraic behavioral equivalence captures weighted bisimilarity (*Klin*, 2009). Concretely, a weighted bisimulation is an equivalence relation $R \subseteq C \times C$ such that for all $x \mathrel{R} y$ and $z \in C$:*

$$\sum_{z \mathrel{R} z'} c(x)(z') = \sum_{z \mathrel{R} z'} c(y)(z')$$

5. *Taking $M = (\mathbb{Q}, +, 0)$, we get that $M^{(X)}$ are linear combinations over X. If we restrict to the subfunctor $\mathcal{D}(X) = \{f \in \mathbb{Q}_{\geq 0}^{(X)} \mid \sum_{x \in X} f(x) = 1\}$ where the weights are nonnegative and sum to 1, we get (rational finite support) probability distributions over X.[1]*

---

[1] In models of computation where addition of rational numbers isn't linear time, one can restrict to fixed-precision rationals $\mathbb{Q}_q = \{\frac{p}{q} \mid p \in \mathbb{Z}\}$ for some fixed $q \in \mathbb{N}_{>0}$ to obtain our time complexity bound.

Table 1: List of functors, their coalgebras, and the accompanying notion of behavioral equivalence. The first five is given in Theorem 8.3.3, the last introduced later in Section 8.5.

Functor $F(X)$	Coalgebras $c\colon C \to FC$	Coalgebraic equivalence
$\mathcal{P}_f(X)$	Transition Systems	(Strong) Bisimilarity
$\mathcal{P}_f(A \times X)$	Labelled Transition Systems	(Strong) Bisimilarity
$M^{(X)}$	Weighted Systems (for a monoid M)	Weighted Bisimilarity
$\mathcal{P}_f(\mathcal{D}(X))$	Markov Decision Processes	Probabilistic Bisimilarity
$M^{(\tilde{\Sigma}X)}$	Weighted Tree Automata	Backwards Bisimilarity
$\mathcal{N}(X)$	Monotone Neighbourhood Frames	Monotone Bisimilarity

6. *For two functors* F *and* G*, we can consider the coalgebra over their composition* F ∘ G*. Taking* F = $\mathcal{P}_f$ *and* G = A × (−)*, coalgebras over* F ∘ G *are labelled transition systems with strong bisimilarity. Taking* F = $\mathcal{P}_f$ *and* G = $\mathcal{D}$*, coalgebras over* F ∘ G *are Markov decision processes with probabilistic bisimilarity (Larsen and Arne Skou, 1991, Def. 6.3), (Bartels et al., 2003, Thm. 4.2). For* F = M$^{(−)}$ *and* G = Σ *for some signature functor,* FG-*coalgebras are weighted tree automata and coalgebraic behavioral equivalence is backward bisimilarity (Deifel et al., 2019; Björklund et al., 2009).*

Sometimes, we need to reason about successors and predecessors of a general F-coalgebra:

Definition 8.3.4. Given a coalgebra $c\colon C \to FC$ and a state $x \in C$, we say that $y \in C$ is a *successor of* x if $c(x)$ is not in the image of $Fi_y\colon F(C \setminus \{y\}) \to FC$, where $i_y\colon C \setminus \{y\} \rightarrowtail C$ is the canonical inclusion. Likewise, x is a *predecessor* of y, and the *outdegree* of x is the number of successors of x.

Intuitively, y is a successor of x if y appears somewhere in the term that defines $c(x) \in F(C)$, like we did in the "coalgebra" row in Figure 45. We will access the predecessors in the minimization algorithm, and moreover, the total and maximum number of successors will be used in the run time complexity analysis.

8.4 COALGEBRAIC PARTITION REFINEMENT

In this section we will describe how the coalgebraic notions of the preceding section can be used for automata minimization.

8.4.1 *Representing Abstract Data*

When writing an abstract algorithm, it is crucial for the complexity analysis, how the abstract data is actually represented in memory. We understand finite sets like

the carrier of the input coalgebra as finite cardinals $C \cong \{0, \ldots, |C| - 1\} \subseteq \mathbb{N}$, and a map $f\colon C \to D$ for finite C is represented by an array of length $|C|$.

COALGEBRA IMPLEMENTATION   The coalgebra $c\colon C \to FC$ that we wish to minimize is given to the algorithm as a black-box, because it only needs to interact with the coalgebra via a specific interface. Whenever the algorithm comes up with a partition $p\colon C \to C'$, two states $x, y \in C$ need to be moved to different blocks if $F[p](c(x)) \neq F[p](c(y))$. Hence, the algorithm needs to derive $F[p](c(x))$ for states of interest $x \in C$. Since all partitions are finite, we can assume $C' \subseteq \mathbb{N}$, and so for simplicity, we consider partitions as maps $p\colon C \to \mathbb{N}$ with the image $\mathbb{I}(p) = \{0, \ldots, |C'| - 1\}$ and so $F[p](c(x))$ is an element of the set $F\mathbb{N}$.

For the case of labelled transition systems, i.e. $F(X) = \mathcal{P}_f(A \times X)$, the binary representation of $F[p](c(x))$ is called the *signature of $x \in C$ with respect to* p (Blom and Orzan, 2005). This straightforwardly generalizes to arbitrary functors F (Birkmann et al., 2022; Wißmann et al., 2020), so we reuse the terminology *signature* for the binary encoding of the successor structure of $x \in C$ with respect to the blocks the partition p of the previous iteration.

Beside the signatures, the optimized minimization algorithm needs to be able to determine the predecessors of a state, in order to determine which states to mark dirty. Formally, we require:

Definition 8.4.1. The *implementation* of an F-coalgebra $c\colon C \to FC$ is the data $(n, \text{sig}, \text{pred})$ where:

1. $n \in \mathbb{N}$ is a natural number such that $C \cong \{0, \ldots, n - 1\}$

2. $\text{sig}\colon C \times (C \to \mathbb{N}) \to 2^*$ is a function that given a state and a partition, computes the successor structure of the state (represented a binary data), satisfying for all partitions $p\colon C \to \mathbb{N}$ (encoded as an array of size $|C|$) that

$$\forall x, y \in C: \qquad \text{sig}(x, p) = \text{sig}(y, p) \quad \Leftrightarrow \quad F[p](c(x)) = F[p](c(y)) \qquad (4)$$

3. $\text{pred}\colon C \to \mathcal{P}_f C$ is a function such that $\text{pred}(x)$ contains the predecessors of x.

Passing such a general interface makes the algorithm usable as a library, because the coalgebra can be represented in an arbitrary fashion in memory, as long as the above functions can be implemented.

The equivalence involving sig (4) specifies that the binary data of type 2^* returned by sig is some normalized representation of $F[p](c(x)) \in F\mathbb{N}$. For example, in the implementation for $F = \mathcal{P}_f$, an element of $F\mathbb{N} = \mathcal{P}_f\mathbb{N}$ is a set of natural numbers. Since e.g. $\{2, 0\}$ and $\{0, 2, 2\} \in \mathcal{P}_f\mathbb{N}$ are the same set, the sig function essentially needs to sort the arising sets and remove duplicates:

Example 8.4.2. *We can represent $\mathcal{P}_f$-coalgebras $c\colon C \to \mathcal{P}_f C$ by keeping for every state $x \in C$ an array of its successors $c(x) \subseteq C$ in memory. As a pre-processing step, we directly*

Algorithm 3 Renumbering an array using radix sort

---

procedure RENUMBER(p: B → 2*)
    Create a new array r of size |B| containing numbers 0..|B|
    Sort r by the key p: B → 2* using radix-sort
    Create a new array p′: B → ℕ
    j ← 0
    for i ∈ 0..|B| **do**
        if i > 0 and p[r[i − 1]] ≠ p[r[i]] **then** j ← j + 1
        p′[r[i]] ← j
    return p′

---

compute the predecessors for each state $x \in C$ and keep them as an array $\mathsf{pred}(x) \subseteq C$ for every state x in memory as well (computing the predecessors of all states can be done in linear time, and thus does not affect the complexity of the algorithm). With $n := |C|$, the remaining function sig is implemented as follows:

1. *Given $p: C \to \mathbb{N}$ and $x \in C$, create a new array t of integers of size $|c(x)|$. For each successor $y \in c(x)$, add $p(y) \in \mathbb{N}$ to t; this runs linearly in the length of t because we assume that the map p is represented as an array with $O(1)$ access.*

2. *Sort t via radix sort and then remove all duplicates, with both steps taking linear time.*

3. *Return the binary data blob of the integer array t.*

For $\mathcal{P}_f$, the computation of the signature of a state $x \in C$ thus takes $O(|c(x)|)$ time.

We discuss further instances in Section 8.5 later.

RENUMBER   By encoding everything as binary data in a normalized way, we are able to make heavy use of radix sort, and thus achieve linear bounds on sorting tasks. This trick is also used in the complexity analysis of Kanellakis and Smolka, who refer to it as lexicographic sorting method by Aho, Hopcroft, and Ullman (Aho et al., 1974). We use this trick in order to turn arrays of binary data $p: B \to 2^*$ into their corresponding partitions $p': B \to \{0, \ldots, |\Uparrow(p)| - 1\}$ satisfying $p(x) = p(y) \iff p'(x) = p'(y)$ for all $x, y \in B$. The pseudocode is listed in Algorithm 3: first, a permutation $r: B \to B$ is computed such that $p \circ r: B \to 2^*$ is sorted. This radix sort runs in $O(\sharp(p))$, where $\sharp(p) = \sum_{x \in B} |p(x)|$ is the total size of the entire array p. Since identical entries in p are now adjacent, a simple for-loop iterates over r and readily assigns block numbers.

Lemma 8.4.3. *Algorithm 3 runs in time $O(\sharp(p))$ for the parameter $p: B \to 2^*$ and returns a map $p': B \twoheadrightarrow b$ for some $b \in \mathbb{N}$ such that for all $x, y \in B$ we have $p(x) = p(y) \Leftrightarrow p'(x) = p'(y)$.*

In the actual implementation, we use hash maps to implement RENUMBER. This is faster in practice but due to the resolving of hash-collisions, the theoretical worst-case complexity of the implementation has an additional log factor.

The renumbering can be understood as the compression of a map $p\colon B \to 2^*$ to an integer array $p'\colon B \to \mathbb{N}$. In the algorithm, the array elements of type 2^* are encoded signatures of states.

8.4.2 *The Naive Method Coalgebraically*

To illustrate the use of the encoding and notions defined above, let us restate the naive method (Algorithm 1, (König and Küpper, 2014; Kanellakis and Smolka, 1983)) in Algorithm 4. Recall that the basic idea is that it computes a sequence of partitions $p_i\colon C \to P_i$ ($i \in \mathbb{N}$) for a given input coalgebra $c\colon C \to FC$. Initially this partition identifies all states $p_0\colon C \to 1$. In the first iteration, the map $p'\colon (C \xrightarrow{c} FC \xrightarrow{F[p]} F\mathbb{N})$ sends each state to its *output behavior* (this distinguishes final from non-final states in DFAs and deadlock from live states in transition systems). Then this partition is refined successively under consideration of the transition structure: x, y are identified by $p_{i+1}\colon C \to P_{i+1}$ iff they are identified by the composed map

$$C \xrightarrow{c} FC \xrightarrow{F[p_i]} FP_i.$$

The algorithm terminates as soon as $p_i = p_{i+1}$, which then identifies precisely the behaviorally equivalent states in the input coalgebra (C, c).

Algorithm 4 The naive algorithm, also called *final chain partitioning*

---

  procedure NAIVEALGORITHM′$(c\colon C \to FC)$
    Create a new array $p\colon C \to \mathbb{N} := (x \mapsto 0)$ ▷ *i.e.* $p[x] = 0$ *for all* $x \in C$
    while $|\mathbb{1}(p)|$ changes **do**
      compute $p'\colon C \to 2^* := x \mapsto \mathrm{sig}(x, p)$ ▷ $p'[x] \in 2^*$ *is the encoding of*
      $F[p](c(x)) \in F\mathbb{N}$
      $p\colon C \to \mathbb{N} \leftarrow$ RENUMBER(p')

Recently, Birkmann et al. (Birkmann et al., 2022) have adapted this algorithm to a distributed setting, with a run time in $O(m \cdot n)$.

8.4.3 *The Refinable Partition Data Structure*

For the naive method it sufficed to represent the quotient on the state space $p\colon C \to \mathbb{N}$ by a simple array. For more efficient algorithms like our Algorithm 2, it is crucial to quickly perform certain operations on the partition, for which we have built upon a refinable partition data structure (Valmari, 2009; Valmari and Lehtinen, 2008). The data structure keeps track of the partition of the states into blocks. A key requirement for our algorithm is the ability to split a block into k sub-blocks, where k is arbitrary. The refinable partition also tracks for each state whether it is *clean* or *dirty*, and a *worklist* of blocks with at least one dirty state.

Let us define the exposed functionality of the refinable partition data structure:

1. Given (the natural number identifying) a block B, return its dirty states B_{di} in $O(|B_{di}|)$.

2. Given a block B, return one arbitrary clean state in $O(1)$ if there is any. We denote this by the set B_{cl_1} of cardinality at most 1. B_{cl_1} contains a clean state of B or is empty if all states of B are dirty.

3. Return an arbitrary block with a dirty state and remove it from the worklist, in $O(1)$.

4. MARKDIRTY(s): mark state s dirty, and put its block on the worklist, in $O(1)$.

5. SPLIT(B, A): split a block B into many sub-blocks according to an array $A: B_{di} \to \mathbb{N}$. The array A indicates that the i-th dirty state is placed in the sub-block $A[i]$, meaning that two states s_1, s_2 stay together iff $A[s_1] = A[s_2]$. The clean states are placed in the 0-th sub-block, with those states satisfying $A[s] = 0$.

   The block identifier of B gets re-used as the identifier for largest sub-block, and all states of B are marked clean. SPLIT returns the list of all newly allocated sub-blocks, i.e. those except the re-used one.

   For the time complexity of our algorithm, it is important that SPLIT(B, A) runs in time $O(|B_{di}|)$, regardless of the number of clean states.

In order to implement these operations with the desired run time complexity, we maintain the following data structures:

- `loc2state` is an array of size $|C|$ containing all states of C. Every block is a section of this array, and the other stuctures are used to quickly find and update the entries in the `loc2state` array. A visualization of an extract of this array is shown in Algorithm 5; for example lowermost row shows three blocks of size 5, 3, and 1, respectively.

- The array `state2loc` is inverse to `loc2state`; `state2loc[s]` provides the index ("location") of state s in `loc2state`.

- `blocks` is an array of tuples $(start, mid, end)$ and specifies the blocks of the partition. A block identifier B is simply an index in this array and `blocks[B] = (start, mid, end)` means that block B starts at `loc2state[start]` and ends before `loc2state[end]`, as indicated in the visualization in Algorithm 5. The range start..mid contains the clean states of B and mid..end the dirty states. E.g. mid = end iff the block has no dirty states.

- The array `block_of` of size $|C|$ that maps every state $s \in C$ to the ID B = `block_of[s]` of its surrounding block.

- `worklist` is a list of block identifiers and mentions those blocks with at least one dirty state.

Algorithm 5 Refinable partition data structure with n-way split

procedure MARKDIRTY(s)
  ▷ *Determine the block data* ◁
  B := block_of[s]
  j := state2loc[s]
  (start, mid, end) := blocks[B]
  ▷ *Do nothing if already dirty* ◁
  if mid ≤ j **then return**
  ▷ *Add to worklist if first dirty state* ◁
  if mid = end **then**
  worklist.add(B)
  ▷ *Swap s with the last clean state* ◁
  s' := loc2state[mid − 1]
  state2loc[s'] := j
  state2loc[s] := mid
  loc2state[j] := s'
  loc2state[mid] := s
  ▷ *Move marker to make s dirty* ◁
  blocks[B].mid −= 1
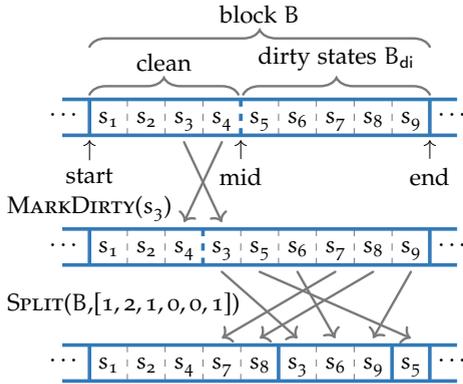
procedure SPLIT(B, A: $B_{di} \to \mathbb{N}$)
  ▷ *Cumulative counts of sub-block sizes* ◁
  (start, mid, end) := blocks[B]
  $D[0.. \max_i A[i] + 1] := 0$
  D[0] := mid − start
  for j ∈ B_{di} **do**
    D[A[j]] += 1
  $i_{max} = \text{argmax}_i D[i]$
  for i ∈ 1..|D| **do**
    D[i] += D[i − 1]
  ▷ *Re-order the states by A-value* ◁
  dirty := copy(loc2state[mid..end])
  for i ∈ reverse(0..|A|) **do**
    D[A[i]] −= 1
    j := start + D[A[i]]
    loc2state[j] := dirty[i]
    state2loc[loc2state[i]] := j
  D[0] −= mid − start
  ▷ *Create blocks and assign IDs* ◁
  D.add(end − start)
  old_block_count := |blocks|
  for i ∈ 0..|D| − 1 **do**
    j_0 := start + D[i]
    j_1 := start + D[i + 1]
    if i = i_{max} **then**
      blocks[B] = (j_0, j_1, j_1)
    else
      blocks.add(j_0, j_1, j_1)
      idx := |blocks| − 1
      block_of[loc2state[$j_0..j_1$]] := idx
  return old_block_count..|blocks|

With this data, we can implement the above-mentioned interface:

1. For a block B, its dirty states B_{di} are the states loc2state[mid..end] where blocks[B] = (start, mid, end).

2. One arbitrary clean state B_{cl_1} of a given block B is determined in a similar fashion: for blocks[B] = (start, mid, end), if start = mid, then there is no clean state B_{cl_1} = {}, and otherwise we chose B_{cl_1} = {loc2state[start]}.

3. Returning an arbitrary block containing a dirty state is just a matter of extracting one element from `worklist`.

4. The pseudocode of MARKDIRTY is listed in Algorithm 5: when marking a state $s \in C$ dirty, we first find the boundaries $(start, mid, end) = \text{blocks}[B]$ of the surrounding block $B = \text{block\_of}[s]$. By the index `state2loc[s]`, we can check in $O(1)$ whether s is in the first ("clean") or second ("dirty") part of the block. Only if s wasn't dirty already, we need to do something: if B did not contain dirty states yet ($start = mid$), B now needs to be added to the `worklist`. Then, we change the location of s in the main array such that it becomes the last clean state, and then we make it dirty by decrementing the index mid.

In the example in Algorithm 5, the content of `loc2state` is visualized. The bold dashed line visualizes the mid position, so states on the left of it are clean, states on the right are dirty. The call to MARKDIRTY(s_3) transforms the first row into the second row: it does so by moving s_3 from the clean states of B to the dirty ones, while s_4 stays clean.

5. The pseudocode of SPLIT is listed in Algorithm 5: for a block B, the caller provides us with an array $A: B_{di} \to \mathbb{N}$ that specifies which of the states stay together and which are moved to separate blocks. In the visualized example, $A = [1, 2, 1, 0, 0, 1]$ represents the map

$$s_3 \mapsto 1, \quad s_5 \mapsto 2, \quad s_6 \mapsto 1, \quad s_7 \mapsto 0, \quad s_8 \mapsto 0, \quad s_9 \mapsto 1$$

So SPLIT(B, A) needs to create new blocks s_3, s_6, s_9 and s_5, while s_7, s_8 stay with the clean states. In any case, the clean states stay in the same block, so we can understand A as an efficient representation of the map

$$\bar{A}: B \to \mathbb{N} \qquad \bar{A}(s) = \begin{cases} A(s) & \text{if } s \in B_{di}, \\ 0 & \text{otherwise.} \end{cases}$$

Then, two states $s, s' \in B$ stay in the same block iff $\bar{A}[s] = \bar{A}[s']$. In the implementation, we first create an auxiliary array D which has different meanings. Before the definition of i_{max}, it counts the sizes of the resulting blocks:

$$D[i] = \{j \in B \mid \bar{A}[j] = i\}.$$

We compute D by initializing $D[0]$ with the number of clean states ($mid - start$) and iterating over A. The index of the largest block remembered in i_{max}, and then we change the meaning of D such that it now holds partial sums $D[i] := \sum_{0 \leqslant j < i} D[j]$. For every new block i, this sum $D[i]$ denotes the end of the block, relative to the start of the old block B.

We use the sums to re-order the states such that states belonging to the same sub block come next to each other. The for-loop moves every state $i \in B$ to

the end of the new block $A[i]$ and decrements $D[A[i]]$ such that the next state belonging to $A[i]$ is inserted before that. Finally, we do not need to move the clean states to sub-block 0, so we simply decrement $D[0]$ by the number of clean states. Since we have inserted all the elements at the end of their future subblocks and have decremented the entry of D during each insertion, the entries of D now point to the *first element* of each future subblock.

Having the states in the right position within B, we can now create the subblocks with the right boundaries. For convenience, we add the (relative) end of B to D, because then, every sub block i ranges from $D[i]$ to $D[i + 1]$. We had saved the index of the largest subblock i_{max}, which will inherit the block identifier of B and the entry blocks[B]. For all other subblocks, we add a new block to blocks. All new blocks have no dirty states, so mid = end for the new entries. If we have added a new block, then we need to update block_of[s] for every state s in the subblock.

8.4.4 *Optimized Algorithm*

With the refinable partition data structure at hand, we can improve on the naive algorithm without restricting the choice of F. Our efficient algorithm is given in . We start by creating a refinable partition data structure with a single block for all the states. We then iterate while there is still a block with dirty states, i.e. with states whose signatures should be recomputed. We split the block into sub-blocks in a refinement step that is similar to the naive algorithm, and re-use the old block for the largest sub-block.

To achieve our complexity bound, this splitting must happen in time $|B_{di}|$, regardless of the number of clean states. Fortunately, this is possible because the clean states all have the same signature, because all their successors remained unchanged. Hence, it suffices to compute the signature for one arbitrary clean state, denoted by B_{cl_1}. Depending on the functor, it might happen that there are dirty states $d \in B_{di}$ that have the same signature as the clean states. Having marked a state as "dirty" just means that the signature might have changed compared to the previous run, so it might be that the signature of a dirty state turns out to be identical to the clean states in the block B.

The wrapper RENUMBER′ then first compresses $p: B_{di} \to 2^*$ to $A: B_{di} \to \mathbb{N}$. Then, RENUMBER′ ensures that those dirty states $d \in B_{di}$ with the same signature as the clean states satisfy $A(d) = 0$. This is used in SPLIT: in the splitting operation, two dirty states $d, d' \in B_{di}$ stay in the same block iff $A(d) = A(d')$ and the clean states end up in the same block as the dirty states d with $A(d) = 0$.

After the block B is split, we need to mark all states $x \in B$ as dirty whose signature might have possibly changed due to the updated partition. If the successor y of $x \in B$ was moved to a new block, i.e. if $p(y)$ changed, this might affect the signature

---

Algorithm 6 Optimized Partition Refinement for all Set functors

---

procedure PARTREFSETFUN(C, sig, pred)   ▷ *i.e. for the implementation of* $c \colon C \to FC$
  Create a new refinable partition structure $p \colon C \to \mathbb{N}$
  Init p to have one block of all states, and all states marked dirty.
  while there is a block B with a dirty state **do**

  Compute the arrays   ⎫ Compute signatures,
    $(sigs_{di} \colon B_{di} \to 2^*) := (x \mapsto sig(x, p))$ ⎬ in total $O(m \log n)$ calls
    $(sigs_{cl} \colon B_{cl_1} \to 2^*) := (x \mapsto sig(x, p))$ ⎭ to the coalgebra

    $A \colon B_{di} \to \mathbb{N} := \text{RENUMBER}'(sigs_{di}, sigs_{cl})$ ⎫ Split B according to
    $\vec{B}_{new} := \text{SPLIT}(B, A)$ ⎬ signatures in $O(|B_{di}|)$

    for every $B' \in \vec{B}_{new}$ and $s \in B'$ **do**   ⎫ Mark dirty all states with
      for every $s' \in pred(s)$ **do**   ⎬ a successor in a new block
        MARKDIRTY(s')   ⎭ in total time $O(m \log n)$
  return the partition p
procedure RENUMBER$'(p \colon B_{di} \to 2^*, q \colon B_{cl_1} \to 2^*)$
  $(A \colon B_{di} \to \mathbb{N}) := \text{RENUMBER}(p)$
  if $d \in B_{di}, c \in B_{cl_1}$ with $p(d) = q(c)$ **then**   ⎫ Ensure A(d)=0 for all
    Swap the values 0 and A(d) in array A   ⎬ states d with the same
  return A   ⎭ signature as clean states.

---

of x. Conversely, if no successor of x changed block, then the signature of x remains unchanged:

Lemma 8.4.4. *If for a finite coalgebra* $c \colon C \to FC$, *two partitions* $p_1, p_2 \colon C \to \mathbb{N}$ *satisfy* $p_1(y) = p_2(y)$ *for all successors* y *of* $x \in C$, *then* $F[p_1](c(x)) = F[p_2](c(x))$.

We can now prove correctness of the partition refinement for coalgebras:

Theorem 8.4.5. *For a given coalgebra* $c \colon C \to FC$, *Algorithm 6 computes behavioral equivalence.*

8.4.5   *Complexity Analysis*

We structure the complexity analysis as a series of lemmas phrased in terms of the number of states $n = |C|$ and the total number of transitions m defined by

$$m := \sum_{x \in C} |pred(x)|$$

As a first observation, we exploit that SPLIT re-uses the block index for the largest resulting block. Thus, whenever x is moved to a block with a different index, the new block has at most half the size of the old block, leading to the logarithmic factor, by Hopcroft's trick:

Lemma 8.4.6. *A state is moved into a new block at most $O(\log n)$ times, that is, for every $x \in C$, the value of $p(x)$ in [Algorithm 6](#) changes at most $\lceil \log_2 |C| \rceil$ many times.*

When a state is moved to a different block, all its predecessors are marked dirty. If there are m transitions in the system, and each state is moved to different block at most $\log n$ times, then:

Lemma 8.4.7. MARKDIRTY *is called at most $m \cdot \lceil \log n \rceil + n$ many times (including initialization).*

In the actual implementation, we arrange the pointers in the initial partition directly such that all states are marked dirty when the main loop is entered for the first time. The overall run time is dominated by the complexity of sig and pred. Here, we assume that sig always takes at least the time needed to write its return value. On the other hand, we allow that pred returns a pre-computed array by reference, taking only $O(1)$ time. The pre-computation of pred can be done at the beginning of the algorithm by iterating over the entire coalgebra once, e.g. it can be done along with input parsing. This runs linear in the overall size of the coalgebra, and thus is dominated by the complexity of the algorithm:

Proposition 8.4.8. *The run time complexity of [Algorithm 6](#) amounts to the time spent in sig and in pred plus $O(m \cdot \log n + n)$.*

Thus, it remains to count how often the algorithm calls sig. Roughly, sig is called for every state that becomes dirty, so we can show:

Theorem 8.4.9. *The number of invocations of sig in [Algorithm 6](#) is bounded by $O(m \cdot \log n + n)$.*

Corollary 8.4.10. *If sig takes f time, if pred runs in $O(1)$ (returning a reference) and $m \geqslant n$, then [Algorithm 6](#) computes behavioral equivalence in the input coalgebra in $O(f \cdot m \cdot \log n)$ time.*

Example 8.4.11. *For $\mathcal{P}_f$-coalgebras, sig takes $O(k)$ time, if every state has at most k successors. Then [Algorithm 6](#) minimizes $\mathcal{P}_f$-coalgebras in time $O(k \cdot m \cdot \log n)$. Note that $m \leqslant k \cdot n$, so the complexity is also bounded by $O(k^2 \cdot n \log n)$.*

8.4.6 *Comparison to Related Work on the Algorithmic Level*

We can classify partition refinement algorithms by their time complexity, and by the classes of functors they are applicable to. For concrete system types, there are more algorithms than we can recall, so instead, we focus on early representatives and on generic algorithms.

THE HOPCROFT LINE OF WORK.   One line of work originates in Hopcroft's 1971 work on DFA minimization (Hopcroft, 1971), and continues with Kanellakis and

Smolka's (Kanellakis and Smolka, 1983, 1990) work on partition refinement for transition systems running in $O(k^2 n \log n)$ where k is the maximum out-degree. It was a major achievement by Paige and Tarjan (Paige and Tarjan, 1987) to reduce the run time to $O(kn \log n)$ by counting transitions and storing these transition counters in a clever way, which subsequently lead to a fruitful line of research on transition system minimization (Garavel and Lang, 2022). This was generalized to coalgebras in Deifel, Dorsch, Milius, Schröder and Wißmann's work on *CoPaR*, which is applicable to a large class of functors satisfying their *zippability* condition. These algorithms keep track of a worklist of blocks with respect to which *other* blocks still have to be split. Our algorithm, by contrast, keeps track of a worklist of blocks that *themselves* still potentially have to be split. Although similar at first sight, they are fundamentally different: in the former, one is given a block, and must determine how to split all the predecessor blocks, whereas in our case one is given a block, which is then split based on its successors.

The advantage of the former class of algorithms is that they have optimal time complexity $O(kn \log n)$, provided one can implement the special splitting procedure for the functor. The additional memory needed for the transition counters is linear in kn.

Our algorithm, by contrast, has an extra factor of k, but is applicable to all computable set-functors. By investing this extra time-factor k, we reduce the memory consumption because we do not need to maintain transition counters or intermediate states like *CoPaR*.

A practical advantage of our algorithm is that *one* recomputation of a block split can take into account the changes to *all* the other blocks that happened since the recomputation. The Hopcroft-*CoPaR* line of work, on the other hand, has to consider each change of the other blocks separately. This advantage is of no help in the asymptotic complexity, because in the worst case only one other split happened each time, and then our algorithm does in $O(k)$ what *CoPaR* can do in $O(1)$. However, as we shall see in the benchmarks of Section 8.6, in practice our algorithm outperforms *CoPaR* and *mCRL2*, even though our algorithm is applicable to a more general class of functors.

THE MOORE LINE OF WORK.   Another line of work originates in Moore's 1956 work on DFA minimization (Moore, 1956), which in retrospect is essentially the naive algorithm specialized to DFAs. In this class, the most relevant for us is the algorithm by König and Küpper (König and Küpper, 2014) for coalgebras, and the distributed algorithm of Birkmann, Deifel, and Milius (Birkmann et al., 2022). Like our algorithm, algorithms in this class split a block based on its successors, and can be applied to general functors. Unlike the Hopcroft-*CoPaR* line of work and our algorithm, the running time of these algorithms is $O(kn^2)$.

Another relevant algorithm in this class is the algorithm of Blom and Orzan (Blom and Orzan, 2005) for transition systems. Their main algorithm runs in time $O(kn^2)$, but in a side note they mention a variation of their algorithm that runs in

$O(n \log n)$ *iterations*. They do not further analyse the time complexity or describe how to implement an iteration, because the main focus of their paper is a distributed implementation of the $O(kn^2)$ algorithm, and the $O(n \log n)$ variation precludes distributed implementation. Out of all algorithms, Blom and Orzan's $O(n \log n)$ variation is the most similar to our algorithm, in particular because their algorithm is in the Moore line of work, yet also re-uses the old block for the largest sub-block (which is a feature that usually appears in the Hopcroft-*CoPaR* line of work). However, their block splitting is different from ours and is only correct for labelled transition systems but can not be easily applied to general functors F.

8.5 INSTANCES

We give a list of examples of instances that can be supported by our algorithm. We start with the instances that were already previously supported by *CoPaR*, and then give examples of instances that were not previously supported by $n \log n$ algorithms.

8.5.1 *Instances also Supported by CoPaR*

PRODUCTS AND COPRODUCTS   The simplest instances are those built using the product $F \times G$ and disjoint union $F + G$, or in general, signature functors for countable signatures Σ. The binary encoding of an element of signature functor $(\sigma, x_1, \ldots, x_k) \in \tilde{\Sigma}X$ starts with a specification of σ, followed by the concatenation of encodings of the parameters $x_1, \ldots, x_k$. The functor implementation can simply apply the substitution recursively to these elements $x_1, \ldots, x_k$, without any further need for normalization.

POWERSET   The finite powerset functor $\mathcal{P}_f$ can be used to model transition systems as coalgebras. In conjunction with products and coproducts, we can model nondeterministic (tree) automata and labelled transition systems. The binary encoding of an element $\{x_1, \ldots, x_k\}$ of the powerset functor, is stored as a list of elements prefixed by its length. The functor implementation can recursively apply the substitution to the elements of a set $\{x_1, \ldots, x_k\}$, and subsequently normalize by sorting the resulting elements and removing adjacent duplicates.

MONOID-VALUED FUNCTORS   The binary encoding of $\mu \in M^{(X)}$ (for a countable monoid M) is an array of pairs $(x_i, \mu(x_i))$. The binary encoding stores a list of these pairs prefixed by the length of the list. The functor implementation recursively applies the substitution to the x_i, and then sorts the pairs by the x_i value, and removes adjacent duplicate x_i by summing up their associated monoid values $\mu(x_i)$.

8.5.2 *Instances not Supported by CoPaR*

COMPOSITION OF FUNCTORS WITHOUT INTERMEDIATE STATES The requirement of zippability in the $m \log n$ algorithm (Deifel et al., 2019) is not closed under the composition of functors $F \circ G$. As a workaround, one can introduce explicit intermediate states between F- and G-transitions. This introduces potentially many more states into the coalgebra, which leads to increased memory usage. Our algorithm can use the composed functor directly without splitting states, because it works for any computable functor. This is important for practical efficiency.

MONOTONE MODAL LOGICS AND MONOTONE BISIMULATION When reasoning about game-theoretic settings (Parikh, 1985; Peleg, 1987; Pauly, 2001), the arising modal logics have modal operators that talk about the ability of agents to enforce properties in the future. This leads to *monotone modal logics* whose domain of reasoning are *monotone neighbourhood frames* and the canonical notion of equivalence is *monotone bisimulation*. It was shown by Hansen and Kupke (Hansen and Kupke, 2004) that these are an instance of coalgebras and coalgebraic behavioral equivalence for the monotone neighbourhood functor. Instead of the original definition, it suffices for our purposes to work with the following equivalent characterization:

Definition 8.5.1 ((Hansen and Kupke, 2004, Lem 3.3)). The monotone neighbourhood functor
$\mathcal{N} \colon \mathsf{Set} \to \mathsf{Set}$ is given by

$$\mathcal{N}X = \{N \in \mathcal{P}_f \mathcal{P}_f X \mid N \text{ upwards closed}\} \text{ and } \mathcal{N}(f \colon X \to Y)(N) = \uparrow\{f[S] \mid S \in N\}.$$

where $\uparrow$ denotes upwards closure.

Hence, in a coalgebra $c \colon C \to \mathcal{N}C$, the successor structure of a state $x \in C$ is an upwards closed family of neighbourhoods $c(x)$.

To avoid redundancy, we do not keep the full neighbourhoods in memory, but only the least elements in this family: given a family $N \in \mathcal{N}X$ for finite X, we define the map

$$\mathsf{atom}_X \colon \mathcal{P}_f \mathcal{P}_f X \to \mathcal{P}_f \mathcal{P}_f X \qquad \mathsf{atom}_X(N) = \{S \in N \mid \nexists S' \in N : S' \subsetneq S\}$$

which transforms a monotone family into an antichain by taking the minimal elements in the monotone family.

Definition 8.5.2. We can implement $\mathcal{N}$-coalgebras as follows: For a coalgebra $c \colon C \to \mathcal{N}C$, keep for every state $x \in C$ an array of arrays representing $\mathsf{atom}_C(c(x)) \in \mathcal{P}_f \mathcal{P}_f X$. The predecessors of a state y needs to be computed in advance and is given by

$$\mathsf{pred}(y) = \{x \in C \mid y \in A \text{ for some } A \in \mathsf{atom}_C(c(x))\}.$$

For the complexity analysis, we specify the out-degree as

$$k := \max_{x \in C} \sum_{S \in \mathrm{atom}_C(c(x))} |S|.$$

For the signature $\mathrm{sig}(x, p)$ of a state x w.r.t. $p \colon C \to \mathbb{N}$, do the following:

1. Compute $\mathcal{P}_f[\mathcal{P}_f[p]](t)$ for $t := \mathrm{atom}_C(c(x))$ by using the sig-implementation of $\mathcal{P}_f$ first for each nested set and then on the outer set. This results in a new set of sets $t' := \mathcal{P}_f[\mathcal{P}_f[p]](t)$.

2. For the normalization, iterate over all pairs $S, T \in t'$ and remove T if $S \subsetneq T$. This step is not linear in the size of t' but takes $O(k^2)$ time.

For such a monotone neighbourhood frame $c \colon C \to \mathcal{N}C$, note that for states $x \in C$, another state $y \in C$ might be contained in multiple sets $S \in c(x)$. Still, the definition of m in the complexity analysis is agnostic of this.

Proposition 8.5.3. *For a monotone neighbourhood frame $c \colon C \to \mathcal{N}C$, let $k \in \mathbb{N}$ be such that $|\mathrm{atom}_C(c(x))| \leqslant k$ for all $x \in C$. Algorithm 6 computes monotone bisimilarity in $O(k^2 \cdot m \log n)$ time.*

8.6 BENCHMARKS

To evaluate the practical performance and memory usage of our algorithm, we have implemented it in our tool *Boa* (Jacobs and Wissmann, 2022), written in Rust. The user of *Boa* can either use a composition of the built-in functors to describe their automaton type, or implement their own automaton type by implementing the interface of Section 8.4.1 in Rust. The user may then input the data of their automaton using either a textual format akin to the representation in the "Coalgebra" row of Figure 45, or use *Boa*'s more efficient and compact binary input format.

We test *Boa* on the benchmark suite of Birkmann, Deifel and Milius (Birkmann et al., 2022), consisting of real-world benchmarks (fms & wlan – from the benchmark suite of the PRISM model checker (Kwiatkowska et al., 2011)), and randomly generated benchmarks (wta – weighted tree automata). For the wta benchmarks, the size of the first 5 was chosen such that *CoPaR* (Deifel et al., 2019) uses 16GB of memory, and the size of the 6th benchmark was chosen by Birkmann, Deifel and Milius to demonstrate the scalability of their distributed algorithm.

The benchmark results are given in Table 2. The first columns list the type of benchmark and the size of the input coalgebra. For the size, the column n denotes the number of states and m is the number of edges as defined in Section 8.4.5. In the wlan benchmarks for CoPaR (Deifel et al., 2019; Wißmann et al., 2021), the reported number of states and eges also include intermediate states introduced by CoPaR in order to cope with functor composition, a preprocessing step which we do not need in *Boa*, and thus are different from the numbers in Table 2 here.

The three subsequent columns list the running time of *CoPaR*, *DCPR*, and *Boa*. The last two columns list the memory usage of *DCPR* and *Boa*. The benchmark results for *DCPR* and *CoPaR* are those reported by Birkmann, Deifel and Milius (Birkmann et al., 2022), and were run on their high performance computing cluster with 32 workers on 8 nodes with two Xeon 2660v2 chips (10 cores per chip + SMT) and 64GB RAM. The memory usage of *DCPR* is *per worker*, indicated by the ×32.

Execution times of *CoPaR* were taken using one node of the cluster. Some entries for *CoPaR* are missing, indicating that it ran out of its 16GB of memory. The benchmark results for our algorithm were obtained on one core of a 2.3GHz MacBook Pro 2019.

A point to note is that compared to *CoPaR*, the distributed algorithm does best on the randomly generated benchmarks. The distributed algorithm beats *CoPaR* in execution time by taking advantage of the large parallel compute power of the HPC cluster. This comes at the cost of $O(n^2)$ worst case complexity, but randomly generated benchmarks are more or less the *best case* for the distributed algorithm, and require only a very small constant number of iterations, so that the effective complexity is $O(n)$. The real world benchmarks on the other hand, and especially the wlan benchmarks, need more iterations, which results in sequential *CoPaR* outperforming *DCPR*. In general, benchmarks with transition systems with long shortest path lengths will truly trigger the worst case of the $O(n^2)$ algorithm, and can make its execution time infeasably long. In summary, the benchmarks here are not chosen to be favourable to *CoPaR* and our algorithm, as they do not trigger the time complexity advantage to the full extent.

Nevertheless, our algorithm outperforms both *CoPaR* and *DCPR* by a large margin. On the synthetic benchmarks (wta), roughly speaking, when *CoPaR* takes 10 minutes, *DCPR* takes one minute, and our algorithm takes a second. On the real-world wlan benchmark, the difference with *DCPR* is greatest, with the largest benchmark requiring almost an hour on the HPC cluster for *DCPR*, whereas our algorithm completes the benchmark in less than a second on a single thread.

Sequential *CoPaR* is unable to run the largest wta benchmarks, because it requires more memory than the 16GB limit. The distributed algorithm is able to spread the required memory usage among 32 workers, thus staying under the 16GB limit per worker. Our algorithm uses sufficiently less memory to be able to run all benchmarks on a single machine. In fact, it uses significantly less memory than *DCPR* uses *per worker*. There are several reasons for this:

- Our algorithm does not require large hash tables.

- Our algorithm uses an binary representation with simple in-memory dictionary compression.

- We operate directly on the composed functor instead of splitting states into pieces.

Even the largest benchmarks stay far away from the 16GB memory limit. We are thus able to minimize large coalgebraic transition systems on cheap, consumer grade hardware.

To assess the cost of genericity, we also compare with *mCRL2*, a full toolset for the verification of concurrent systems. Among many other tasks, *mCRL2* also supports minimization of transition systems by strong bisimilarity as part of the `ltsconvert` command[2] and even implements multiple algorithms for that, out of which the algorithm by Jansen et al. (Jansen et al., 2020) turned out to be the fastest. For benchmarking, we ran its implementation in *mCRL2* and compared the fastest with the run time of *Boa*. As input files, we used the *very large transition systems* (VLTS) benchmark suite[3]. Unfortunately, the benchmark suite is not available online in an open format, so the files were converted with the CADP tool to the plain text `.aut` format, supported by *mCRL2* and our tool. The results are shown in Table 3. The benchmark consists of two series of input files, *cwi* and *vasy*, whose file sizes ranged from a few KB to hundreds of MB (biggest vasy ws 145MB in zipped format and biggest cwi was 630MB zipped). Surprisingly, *Boa* is significantly faster than the bisimilarity minimization implemented in *mCRL2*. On all input files, *mCRL2* and *Boa* agreed on the size of the resulting partition, giving confidence in the correctness of the computed partition. It should be noted that *mCRL2* supports a wide range of bisimilarity notions (e.g. branching bisimilarity), which our algorithm can not cover.

8.7 CONCLUSION AND FUTURE WORK

The coalgebraic approach enables generic tools for automata minimization, applying to different types of input automata. With our coalgebraic partition refinement algorithm, implemented in our tool *Boa*, we reduce the time and memory use compared to previous work. This comes at the cost of an extra factor of k (the outdegree of a state) in the time-complexity compared to asymptotically optimal algorithms. Though our asymptotic complexity is not as good as the asymptotically fastest but less generic algorithms, the evaluation shows the efficiency of our algorithm.

We wish to expand the supported system equivalence notions. So far, our algorithm is applicable to functors on Set. More advanced equivalence and bisimilarity notions such as trace equivalence (Silva and Sokolova, 2011; Hasuo et al., 2007), branching bisimulations, and others from the linear-time-branching spectrum (van Glabbeek, 2001), can be understood coalgebraically using graded monads (Dorsch et al., 2019; Milius et al., 2015), corresponding to changing the base category of the functor from Set to, for example, the Eilenberg-Moore (Silva et al., 2013) or Kleisli (Hasuo et al., 2007) category of a monad. For branching bisimulation, efficient algorithms exist (Jansen et al., 2020; Groote and Vaandrager, 1990), whose ideas might embed into our framework. We conjecture that it is possible to adapt the

---

2 https://www.mcrl2.org/web/user_manual/tools/release/ltsconvert.html
3 https://cadp.inria.fr/resources/vlts/

algorithm to nominal sets, in order to minimize (orbit-)finite coalgebras there (Kozen et al., 2015; Milius et al., 2016; Schröder et al., 2017; Wißmann, 2023).

Up-to techniques provide another successful line of research for deciding bisimilarity. Bonchi and Pous (Bonchi and Pous, 2013) provide a construction for deciding bismilarity of two particular states of interest, where the transition structure is unfolded lazily while the reasoning evolves. By computing the partitions in a similarly lazy way, performance of our minimization algorithm can hopefully be improved even further.

benchmark				time (s)			memory (MB)	
type	n	% red	m	CoPaR	DCPR	Boa	DCPR	Boa
fms	35910	0%	237120	4	2	0.02	$13^{\times 32}$	6
fms	152712	0%	1111482	17	8	0.10	$62^{\times 32}$	20
fms	537768	0%	4205670	68	26	0.40	$163^{\times 32}$	72
fms	1639440	0%	13552968	232	84	1.29	$514^{\times 32}$	199
fms	4459455	0%	38533968	–	406	4.60	$1690^{\times 32}$	557
wlan	248503	56%	437264	39	297	0.11	$90^{\times 32}$	15
wlan	607727	59%	1162573	105	855	0.30	$147^{\times 32}$	38
wlan	1632799	78%	3331976	–	2960	0.81	$379^{\times 32}$	92
$\text{wta}_5(2)$	86852	0%	21713000	537	71	0.85	$701^{\times 32}$	179
$\text{wta}_4(2)$	92491	0%	18498200	723	67	0.96	$728^{\times 32}$	154
$\text{wta}_3(2)$	134207	0%	20131050	689	113	1.34	$825^{\times 32}$	175
$\text{wta}_2(2)$	138000	0%	13800000	467	129	0.98	$715^{\times 32}$	126
$\text{wta}_1(2)$	154863	0%	7743150	449	160	0.74	$621^{\times 32}$	80
$\text{wta}_3(2)$	1300000	0%	195000000	–	1377	22.58	$7092^{\times 32}$	1647
$\text{wta}_5(\text{W})$	83431	0%	16686200	642	52	1.01	$663^{\times 32}$	142
$\text{wta}_4(\text{W})$	92615	0%	23153750	511	61	1.21	$849^{\times 32}$	193
$\text{wta}_3(\text{W})$	94425	0%	14163750	528	59	0.76	$639^{\times 32}$	124
$\text{wta}_2(\text{W})$	134082	0%	13408200	471	76	0.96	$675^{\times 32}$	124
$\text{wta}_1(\text{W})$	152107	0%	7605350	566	79	0.76	$642^{\times 32}$	82
$\text{wta}_3(\text{W})$	944250	0%	141637500	–	675	15.18	$6786^{\times 32}$	1231
$\text{wta}_5(\text{Z})$	92879	0%	18575800	463	56	0.67	$754^{\times 32}$	161
$\text{wta}_4(\text{Z})$	94451	0%	23612750	445	61	0.81	$871^{\times 32}$	199
$\text{wta}_3(\text{Z})$	100799	0%	15119850	391	64	0.62	$628^{\times 32}$	135
$\text{wta}_2(\text{Z})$	118084	0%	11808400	403	74	0.66	$633^{\times 32}$	113
$\text{wta}_1(\text{Z})$	156913	0%	7845650	438	82	0.68	$677^{\times 32}$	93
$\text{wta}_3(\text{Z})$	1007990	0%	151198500	–	645	19.55	$5644^{\times 32}$	1325

Table 2: Time and memory usage comparison on the benchmarks of Birkmann, Deifel and Milius (Birkmann et al., 2022). The columns n, %red, m give the number of states, the percentage of redundant states, and the number of edges, respectively. The results for *Boa* are an average of 10 runs. The results for *CoPaR* and *DCPR* are those reported in Birkmann, Deifel and Milius (Birkmann et al., 2022). The memory usage of *DCPR* is per worker, indicated by ×32 (for the 32 workers on the HPC cluster)

The functors associated with the benchmarks are as follows: **fms:** $F(X) = \mathbb{Q}^{(X)}$, **wlan:** $F(X) = \mathbb{N} \times \mathcal{P}_f(\mathbb{N} \times \mathcal{D}(X))$, **$\text{wta}_r(\text{M})$:** $F(X) = M \times M^{(4 \times X^r)}$ where r indicates the branching factor of the tree automaton, and $M = W$ is the monoid of 64-bit words with bitwise-or, $M = Z$ is the monoid of integers with addition, and $M = 2$ is the monoid of booleans with logical-or.

benchmark				time (s)		memory (MB)	
type	**n**	**% red**	**m**	*mCRL2*	*Boa*	*mCRL2*	*Boa*
cwi	142472	97%	925429	0.85	0.08	99	15
cwi	214202	63%	684419	0.63	0.15	111	16
cwi	371804	90%	641565	0.38	0.11	95	22
cwi	566640	97%	3984157	6.19	0.44	414	60
cwi	2165446	98%	8723465	10.72	1.52	978	166
cwi	2416632	96%	17605592	14.87	1.56	1780	247
cwi	7838608	87%	59101007	231.08	17.43	5777	816
cwi	33949609	99%	165318222	312.11	35.41	16698	2809
vasy	52268	84%	318126	0.31	0.04	48	7
vasy	65537	0%	2621480	6.62	0.14	553	28
vasy	66929	0%	1302664	2.56	0.08	275	18
vasy	69754	0%	520633	0.93	0.04	128	11
vasy	83436	0%	325584	0.38	0.04	86	10
vasy	116456	0%	368569	0.47	0.06	105	15
vasy	164865	99%	1619204	1.92	0.23	162	22
vasy	166464	49%	651168	0.81	0.08	116	16
vasy	386496	99%	1171872	0.67	0.08	133	28
vasy	574057	99%	13561040	18.84	2.41	1277	141
vasy	720247	99%	390999	0.38	0.05	88	31
vasy	1112490	99%	5290860	8.86	0.78	579	93
vasy	2581374	0%	11442382	31.95	2.30	2691	285
vasy	4220790	67%	13944372	31.82	2.87	2293	311
vasy	4338672	40%	15666588	34.89	3.12	3160	372
vasy	6020550	99%	19353474	34.91	4.11	2124	534
vasy	6120718	99%	11031292	15.56	2.37	1297	325
vasy	8082905	99%	42933110	72.45	3.79	4313	719
vasy	11026932	91%	24660513	60.57	6.26	2768	661
vasy	12323703	91%	27667803	63.49	8.16	3103	740

Table 3: Time and memory usage comparison on the VLTS benchmark suite (for space reasons, we have excluded the very short running benchmarks). The columns n, %red, m give the number of states, the percentage of redundant states, and the number of edges, respectively. The results are an average of 10 runs. For *mCRL2*, the default `bisim` option was used, which runs the JGKW algorithm (Jansen et al., 2020).

Part V

CONCLUSION AND FUTURE WORK

Chapter 9

Conclusion and Future Work

Let us conclude this thesis with a summary of the main contributions and a discussion of future work.

PART 1: TYPES FOR DEADLOCK AND LEAK FREE CONCURRENCY

This part of the thesis contributes three new type systems that guarantee deadlock and leak freedom for concurrent programs.

- A language with locks (Chapter 2),

- A language with multiparty message passing (Chapter 3),

- A minimalist language with barriers (Chapter 4).

In each case, the concurrency constructs in these languages are *first-class* or *higher-order*, meaning that they can be passed around as values and stored in data structures, and even recursively applied to themselves.

To prove the soundness of these type systems, we developed a proof method for *deadlock and leak freedom* for such higher-order concurrent programs (Chapter 1).

FUTURE WORK   The type systems presented are for core calculi, and although they support important features such as data types and recursion, they lack many of the convenience features that are present in real-world programming languages. It would be interesting to extend these type systems to more realistic languages, and develop realistic implementations of these languages and type systems. An interesting challenge in this regard is to develop type inference algorithms for these type systems, which might be challenging for the type systems with lock groups and multiparty session types.

Another line of future work is to give a general notion of a deadlock free concurrency construct, to capture the commonalities between constructs such as locks and session-typed channels. This would involve an interface with operations (such as acquire and release for locks, and send and receive for channels), and a set of laws that these operations must satisfy.

Another limitation of connectivity graphs is that they require a 1-to-1 correspondence between physical objects in the run-time configuration, and cgraph objects. This is not always the case in practice. For example, in the case of lock groups, we have to treat the lock group as a single run-time object, even though

it is composed of multiple physical locks. Similarly, in the case of multiparty session types, we have to treat the session as a single run-time object, even though it is composed of multiple physical channels. It would be interesting to extend connectivity graphs to support a 1-to-many correspondence between connectivity graph objects and physical objects.

Lastly, it would be interesting to extend the proof method for deadlock and leak freedom to support stronger results such as liveness and termination. This might possibly be achieved by taking inspiration from from liveness logics such as LiLi (Liang and Feng, 2016) and TaDa Live (D'Osualdo et al., 2021b), and existing work on liveness in Iris (Tassarotti et al., 2017; Spies et al., 2021).

PART 2: SEPARATION LOGICS FOR VERIFIED MESSAGE PASSING

This part of the thesis first contributes a new proof technique for the soundness of separation logics for message passing concurrency (Chapter 5). Secondly, it contributes a new separation logic for message passing concurrency (Chapter 6), which guarantees deadlock and leak freedom. To my knowledge, this is the first separation logic where deadlock and leak freedom follows automatically from linearity in a manner that is analogous to the type systems in Chapters 2 to 4, and hence freely supports higher-order channels.

FUTURE WORK   The primary future challenge for this chapter is to extend the deadlock and leak free separation logic to support other higher-order concurrency constructs. In particular, it would be interesting to extend the separation logic to support locks. Beyond this, it would be interesting to extend the separation logic to support general user-defined concurrency constructs, such as one can do with the Iris separation logic (Jung et al., 2015, 2016, 2018b; Krebbers et al., 2018). In short, we would like to extend Iris with support for deadlock and leak freedom, while retaining the ability to support user-defined higher-order concurrency constructs.

PART 3: PARADOX-FREE PROBABILISTIC PROGRAMMING

This part of the thesis contributes an analysis of the paradoxes that arise in probabilistic programming (Chapter 7). It also contributes a new probabilistic programming language that is free from these paradoxes, using infinitesimal width intervals. In this language, programs are automatically covariant under parameter transformations.

FUTURE WORK   In the future, it would be interesting to integrate the constructs of the language with industrial strength probabilistic programming languages (Carpenter et al., 2017; van de Meent et al., 2018; Bingham et al., 2018; Phan et al.,

2019). On the theory side, it would be interesting to give a more careful account of its semantics along the lines of Staton (2017); Heunen et al. (2017)

Radul and Alexeev (2021) have developed a similar language, but with support for vector-valued random variables. That work was published subsequently to our paper, but developed independently:

> Jacobs (2021) clarifies the scalar-wise treatment by using formal infinitesimals, which notation we adopt in Section 5.2, and also handles conditioning on (unary) transformations of scalar random variables. Our contributions can be viewed as a concurrently-developed extension of the system of Jacobs (2021) to vectorvalued distributions.

It would be interesting to extend our language to support vector-valued distributions, and go further to support other forms of conditioning, for instance, such as in Hakaru (Narayanan et al., 2016).

PART 4: GENERAL AND EFFICIENT AUTOMATON MINIMIZATION

This part of the thesis contributes a new algorithm for automaton minimization (Chapter 8). The algorithm works for a general class of coalgebraic automata specified by a computable Set-functor, and is efficient both in theory and in practice.

FUTURE WORK We wish to expand the supported system equivalence notions. So far, our algorithm is applicable to functors on Set. More advanced equivalence and bisimilarity notions such as trace equivalence (Silva and Sokolova, 2011; Hasuo et al., 2007), branching bisimulations, and others from the linear-time-branching spectrum (van Glabbeek, 2001), can be understood coalgebraically using graded monads (Dorsch et al., 2019; Milius et al., 2015), corresponding to changing the base category of the functor from Set to, for example, the Eilenberg-Moore (Silva et al., 2013) or Kleisli (Hasuo et al., 2007) category of a monad. For branching bisimulation, efficient algorithms exist (Jansen et al., 2020; Groote and Vaandrager, 1990), whose ideas might embed into our framework. We conjecture that it is possible to adapt the algorithm to nominal sets, in order to minimize (orbit-)finite coalgebras there (Kozen et al., 2015; Milius et al., 2016; Schröder et al., 2017; Wißmann, 2023).

Up-to techniques provide another successful line of research for deciding bisimilarity. Bonchi and Pous (Bonchi and Pous, 2013) provide a construction for deciding bismilarity of two particular states of interest, where the transition structure is unfolded lazily while the reasoning evolves. By computing the partitions in a similarly lazy way, performance of our minimization algorithm can hopefully be improved even further.

Part VI

Bibliography

Nathanael L. Ackermann, Cameron E. Freer, and Daniel M. Roy. 2017. On computability and disintegration. *Mathematical Structures in Computer Science* (2017). https://doi.org/10.1017/S0960129516000098

Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph. D. Dissertation. Princeton University.

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* (2010). https://doi.org/10.1145/1709093.1709094

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*.

Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *JCSS* (1989). https://doi.org/10.1007/3-540-19020-1_13

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types*. https://doi.org/10.1007/978-3-319-30936-1_14

Andrew W. Appel. 2014. *Program Logics for Certified Compilers*.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* (2001). https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. https://doi.org/10.1145/1190216.1190235

Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. 2019. Minimal Session Types (Pearl). In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.23

Federico Aschieri, Agata Ciabattoni, and Francesco A. Genco. 2017. Gödel logic: From natural deduction to parallel computation. In *LICS*. https://doi.org/10.1109/LICS.2017.8005076

Arnon Avron. 1991. Hypersequents, Logical Consequence and Intermediate Logics for Concurrency. *Annals of Mathematics and Artificial Intelligence* (1991). https://doi.org/10.1007/BF01531058

Christel Baier, Bettina Engelen, and Mila Majster-Cederbaum. 2000. Deciding Bisimilarity and Similarity for Probabilistic Processes. *J. Comput. Syst. Sci.* (2000). https://doi.org/10.1006/jcss.1999.1683

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. https://doi.org/10.5555/1373322

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. ICFP (2017). https://doi.org/10.1145/3110281

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP*. https://doi.org/10.1007/978-3-030-17184-1_22

Falk Bartels, Ana Sokolova, and Erik de Vink. 2003. A hierarchy of probabilistic system types. In *Coagebraic Methods in Computer Science, CMCS 2003*. https://doi.org/10.1016/j.tcs.2004.07.019

Nick Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *CSL*. https://doi.org/10.1007/BFb0022251

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR*. https://doi.org/10.1007/978-3-540-85361-9_33

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2018). https://doi.org/10.5555/3322706.3322734

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed kripke models over recursive worlds. In *POPL*. https://doi.org/10.1145/1926385.1926401

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* (2010). https://doi.org/10.1016/j.tcs.2010.07.010

Fabian Birkmann, Hans-Peter Deifel, and Stefan Milius. 2022. Distributed Coalgebraic Partition Refinement. In *TACAS*. https://doi.org/10.1007/978-3-030-99527-0_9

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing obligations in higher-order concurrent separation logic. POPL (2019). https://doi.org/10.1145/3290378

Johanna (Högberg) Björklund, Andreas Maletti, and Jonathan May. 2007. Bisimulation Minimisation for Weighted Tree Automata. In *Developments in Language Theory, DLT 2007*. https://doi.org/10.1007/978-3-540-73208-2

Johanna (Högberg) Björklund, Andreas Maletti, and Jonathan May. 2009. Backward and forward bisimulation minimization of tree automata. *Theor. Comput. Sci.* (2009). https://doi.org/10.1016/j.tcs.2009.03.022

Stefan Blom and Simona Orzan. 2005. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer* (2005). https://doi.org/10.1007/s10009-004-0185-2

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR*. https://doi.org/10.1007/978-3-642-15375-4_12

Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. In *POPL*. https://doi.org/10.1145/2429069.2429124

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR*. https://doi.org/10.1007/978-3-540-28644-8_2

Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *FORTE*. https://doi.org/10.1007/978-3-319-39570-8_6

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR*. https://doi.org/10.1007/978-3-642-15375-4_16

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Math. Struct. Comput. Sci.* (2016). https://doi.org/10.1017/S0960129514000218

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *JAR* (2018). https://doi.org/10.1007/s10817-018-9457-5

Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing Order to the Separation Logic Jungle. In *APLAS*. https://doi.org/10.1007/978-3-319-71237-6_10

Marco Carbone and Søren Debois. 2010. A Graphical Approach to Progress for Structured Communication in Web Services. In *ICE*. https://doi.org/10.4204/EPTCS.38.4

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2015.412

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* (2017). https://doi.org/10.1007/s00236-016-0285-y

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* (2017). https://doi.org/10.18637/jss.v076.i01

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. https://doi.org/10.1145/3453483.3454041

David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *TACAS*. https://doi.org/10.1007/978-3-030-45237-7_17

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* (1989). https://doi.org/10.1109/69.43410

Joseph Chang and David Pollard. 1997. Conditioning as disintegration. *Statistica Neerlandica* (1997). https://doi.org/10.1111/1467-9574.00056

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). ICFP (2020). https://doi.org/10.1145/3408998

Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. 2019. Formalized Meta-Theory of Sequent Calculi for Linear Logics. *Theoretical Computer Science* (2019). https://doi.org/10.1016/j.tcs.2019.02.023

Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. 2022. Ferrite: A Judgmental Embedding of Session Types in Rust. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2022.22

Adam Chlipala. 2013. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP*. https://doi.org/10.1145/2500365.2500592

Chromium. 2020. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/

Luca Ciccone and Luca Padovani. 2020. A Dependently Typed Linear π-Calculus in Agda. In *PPDP*. https://doi.org/10.1145/3414080.3414109

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*. https://doi.org/10.1145/286936.286947

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION*. https://doi.org/10.1007/978-3-642-38493-6_4

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS* (2016). https://doi.org/10.1017/S0960129514000188

The Coq-std++ Team. 2021. An extended "standard library" for Coq.

The Coq Team. 2021. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.4501022

Andreea Costea, Wei-Ngan Chin, Shengchao Qin, and Florin Craciun. 2018. Automated Modular Verification for Relaxed Communication Protocols. In *APLAS*. https://doi.org/10.1007/978-3-030-02768-1_16

Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *ICECCS*. https://doi.org/10.1109/ICECCS.2015.33

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *PLDI*. https://doi.org/10.1145/301618.301641

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021a. Certifying Choreography Compilation. In *ICTAC*. https://doi.org/10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Formalising a Turing-Complete Choreographic Language in Coq. In *ITP*. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*. https://doi.org/10.1007/978-3-662-44202-9_9

Fredrik Dahlqvist and Dexter Kozen. 2020. Semantics of higher-order probabilistic programs with conditioning, In POPL. https://doi.org/10.1145/3371125

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. POPL (2020). https://doi.org/10.1145/3371102

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS*. https://doi.org/10.1007/978-3-319-89366-2_5

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *PPDP*. https://doi.org/10.1145/2370776.2370794

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* (2017). https://doi.org/10.1016/j.ic.2017.06.002

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS*. https://doi.org/10.1145/3209108.3209146

Hans-Peter Deifel, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2019. Generic Partition Refinement and Weighted Tree Automata. In *Formal Methods – The Next 30 Years, Proc. 3rd World Congress on Formal Methods (FM 2019)*. https://doi.org/10.1007/978-3-030-30942-8_18

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR*. https://doi.org/10.1007/978-3-642-32940-1_20

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *POPL*. https://doi.org/10.1145/1926385.1926435

Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. 2021. Session Logical Relations for Noninterference. In *LICS*. https://doi.org/10.1109/LICS52264.2021.9470654

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *ESOP*. https://doi.org/10.1007/11785477_20

Edsger W. Dijkstra. 1965. EWD 310: Hierarchical Ordering of Sequential Processes. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

Edsger W. Dijkstra. 1971. Hierarchical Ordering of Sequential Processes. *Acta Informatica* (1971). https://doi.org/10.1007/BF00289519

Ulrich Dorsch, Stefan Milius, and Lutz Schröder. 2019. Graded Monads and Graded Logics for the Linear Time - Branching Time Spectrum. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2019.36

Ulrich Dorsch, Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2017. Efficient Coalgebraic Partition Refinement. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2017.32

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021a. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *TOPLAS* (2021). https://doi.org/10.1145/3477082

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021b. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *TOPLAS* (2021). https://doi.org/10.1145/3477082

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *LMCS* (2011). https://doi.org/10.1109/LICS.2009.34

Thomas Ehrhard. 2018. An Introduction to Differential Linear Logic: Proof-Nets, Models and Antiderivatives. *Math. Struct. Comput. Sci.* (2018). https://doi.org/10.1017/S0960129516000372

Thomas Ehrhard and Laurent Regnier. 2006. Differential Interaction Nets. *Theor. Comput. Sci.* (2006). https://doi.org/10.1016/j.tcs.2006.08.003

Frantisek Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2021. On Algebraic Abstractions for Concurrent Separation Logics. POPL (2021). https://doi.org/10.1145/3434286

Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* (1991). https://doi.org/10.1016/0167-6423(91)90036-W

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2021.36

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types Without Tiers. POPL (2019). https://doi.org/10.1145/3290341

Adrian Francalanza, Julian Rathke, and Vladimiro Sassone. 2011. Permission-Based Separation Logic for Message-Passing Concurrency. *LMCS* (2011). https://doi.org/10.2168/LMCS-7(3:7)2011

Hubert Garavel and Frédéric Lang. 2022. Equivalence Checking 40 Years After: A Review of Bisimulation Tools. https://doi.org/10.1007/978-3-031-15629-8_13

Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* (2005). https://doi.org/10.1007/s00236-005-0177-z

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES*. https://doi.org/10.4204/EPTCS.314.3

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *JFP* (2010). https://doi.org/10.1017/S0956796809990268

Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. 2021. Precise subtyping for asynchronous multiparty sessions. POPL (2021). https://doi.org/10.1145/3434297

Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. 2014. Deadlock Analysis of Unbounded Process Networks. In *CONCUR*. https://doi.org/10.1007/978-3-662-44584-6_6

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. ICFP (2023). https://doi.org/10.1145/3607859

Noah Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *UAI*. https://doi.org/10.5555/2969033.2969207

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

Daniele Gorla. 2010. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation* (2010). https://doi.org/10.1016/j.ic.2010.05.002

Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An Extensible Approach to Session Polymorphism. *MSCS* (2016). https://doi.org/10.1017/S0960129514000231

Jan Friso Groote and Frits W. Vaandrager. 1990. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*. https://doi.org/10.1007/BFb0032063

Jan Friso Groote, Jao Rivera Verduzco, and Erik P. de Vink. 2018. An Efficient Algorithm to Determine Probabilistic Bisimulation. *Algorithms* (2018). https://doi.org/10.3390/a11090131

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. https://doi.org/10.1145/512529.512563

H. Peter Gumm and Tobias Schröder. 2001. Monoid-labeled transition systems. In *Coalgebraic Methods in Computer Science, CMCS 2001*. https://doi.org/10.1016/S1571-0661(04)80908-3

Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP*. https://doi.org/10.1007/978-3-319-89884-1_15

Helle Hvid Hansen and Clemens Kupke. 2004. A Coalgebraic Perspective on Monotone Modal Logic. *Electron. Notes Theor. Comput. Sci.* (2004). https://doi.org/10.1016/j.entcs.2004.02.028

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). https://doi.org/10.5555/3002812

Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. 2007. Generic Trace Semantics via Coinduction. *LMCS* (2007). https://doi.org/10.2168/LMCS-3(4:11)2007

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *LICS*. https://doi.org/10.1109/LICS.2017.8005137

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. POPL (2020). https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *LMCS* (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. https://doi.org/10.1145/3437992.3439914

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* (1969). https://doi.org/10.1145/363235.363259

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*. https://doi.org/10.1007/978-3-540-78739-6_27

Thomas Hofweber. 2014. Infinitesimal Chances. *Philosophers' Imprint* (2014).

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang. (OOPSLA)* (2020). https://doi.org/10.1145/342820

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP*. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* (2016). https://doi.org/10.1145/2827695

John Hopcroft. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*.

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE*. https://doi.org/10.1007/978-3-662-54494-5_7

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *ECOOP*. https://doi.org/10.1007/978-3-540-70592-5_22

Atsushi Igarashi and Naoki Kobayashi. 1997. Type-Based Analysis of Communication for Concurrent Programming Languages. In *SAS*. https://doi.org/10.1007/BFb0032742

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL*. https://doi.org/10.1145/360204.360215

Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the Pi-calculus. *Theoretical Computer Science* (2004). https://doi.org/10.1016/S0304-3975(03)00325-6

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual session types. ICFP (2017). https://doi.org/10.1145/3110282

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-Ocaml: A Session-Based Library with Polarities and Lenses. *Science of Computer Programming* (2019). https://doi.org/10.1016/j.scico.2018.08.005

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session Type Inference in Haskell. In *PLACES*. https://doi.org/10.4204/EPTCS.69.6

Jules Jacobs. 2020. Paradoxes of Probabilistic Programming: Artifact. https://doi.org/10.5281/zenodo.4075076

Jules Jacobs and Stephanie Balzer. 2023. Higher-Order Leak and Deadlock Free Locks. *Proc. ACM Program. Lang.* POPL (2023). https://doi.org/10.1145/3571229

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2021. Appendix and Coq mechanization of "Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic".

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022a. Connectivity graphs: a method for proving deadlock freedom based on separation logic. POPL (2022). https://doi.org/10.1145/3498662

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022b. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.* ICFP (2022). https://doi.org/10.1145/3547638

Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2023. Dependent Session Protocols in Separation Logic from First Principles (Archived Artifact). https://zenodo.org/record/7993904

Jules Jacobs and Thorsten Wissmann. 2022. Boa: binary coalgebraic partition refinement. https://doi.org/10.5281/zenodo.7150706

David N. Jansen, Jan Friso Groote, Jeroen J. A. Keiren, and Anton Wijs. 2020. An O(m log n) algorithm for branching bisimilarity on labelled transition systems. In *TACAS*. https://doi.org/10.1007/978-3-030-45237-7_1

Edwin Thompson Jaynes. 2003. *Probability theory: The logic of science*.

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015a. Session Types for Rust. In *WGP*. https://doi.org/10.1145/2808098.2808100

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015b. Session types for Rust. In *ICFP*. https://doi.org/10.1145/2808098.2808100

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *ESOP*. https://doi.org/10.1007/978-3-642-28869-2_20

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language*. Ph. D. Dissertation. Universität des Saarlandes.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. POPL (2018). https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Paris C. Kanellakis and Scott A. Smolka. 1983. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. https://doi.org/10.1145/800221.806724

Paris C. Kanellakis and Scott A. Smolka. 1990. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Inf. Comput.* (1990). https://doi.org/10.1016/0890-5401(90)90025-D

Ioannis T. Kassios and Eleftherios Kritikos. 2013. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_10

Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, and David Jansen. 2007. Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking. In *TACAS*. https://doi.org/10.1007/978-3-540-71209-1

Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2021. Session Coalgebras: A Coalgebraic View on Session Types and Communication Protocols. https://doi.org/10.1007/978-3-030-72019-3_14

Bartek Klin. 2009. Structural Operational Semantics for Weighted Transition Systems. In *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday*.

Donald E. Knuth. 1965. On the Translation of Languages from Left to Right. *Inf. Control.* (1965). https://doi.org/10.1016/S0019-9958(65)90426-2

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *LICS*. https://doi.org/10.1109/LICS.1997.614941

Naoki Kobayashi. 2002a. A Type System for Lock-Free Processes. *I&C* (2002). https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi. 2002b. Type Systems for Concurrent Programs. https://doi.org/10.1007/978-3-540-40007-3_26

Naoki Kobayashi. 2005. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informatica* (2005). https://doi.org/10.1007/s00236-005-0179-x

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR*. https://doi.org/10.1007/11817949_16

Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock Analysis of Unbounded Process Networks. *Inf. Comput.* (2017). https://doi.org/10.1016/j.ic.2016.03.004

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *TOPLAS* (1999). https://doi.org/10.1145/330249.330251

Naoki Kobayashi, Shin Saito, and Eijiro Sumii. 2000. An Implicitly-Typed Deadlock-Free Process Calculus. In *CONCUR*. https://doi.org/10.1007/3-540-44618-4_35

Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *ICE*. https://doi.org/10.4204/EPTCS.304.4

Wen Kokke and Ornela Dardha. 2021a. Deadlock-Free Session Types in Linear Haskell. In *Haskell Symposium*. https://doi.org/10.1145/3471874.3472979

Wen Kokke and Ornela Dardha. 2021b. Deadlock-free session types in linear Haskell. In *Haskell Symposium*. https://doi.org/10.1145/3471874.3472979

Wen Kokke and Ornela Dardha. 2021c. Prioritise the Best Variation. In *FORTE*. https://doi.org/10.1007/978-3-030-78089-0_6

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: a Fully-Abstract Semantics for Classical Processes. POPL (2019). https://doi.org/10.1145/3290337

Barbara König and Sebastian Küpper. 2014. Generic Partition Refinement Algorithms for Coalgebras and an Instantiation to Weighted Automata. In *Theoretical Computer Science, IFIP TCS 2014*. https://doi.org/10.1007/978-3-662-44602-7

Dexter Kozen, Konstantinos Mamouras, Daniela Petrisan, and Alexandra Silva. 2015. Nominal Kleene Coalgebra. In *Automata, Languages, and Programming, ICALP 2015*. https://doi.org/10.1007/978-3-662-47666-6

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. ICFP (2018). https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-642-22110-1_47

Kim Guldstrand Larsen and Arne Arne Skou. 1991. Bisimulation through Probabilistic Testing. *Inform. Comput.* (1991). https://doi.org/10.1016/0890-5401(91)90030-6

Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. 2013. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*. https://doi.org/10.1007/978-3-319-02444-8_21

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *ESOP*. https://doi.org/10.1007/978-3-642-11957-6_22

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. https://doi.org/10.1145/1111037.1111042

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *POPL*. https://doi.org/10.1145/2837614.2837635

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP*. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016a. Embedding session types in Haskell. In *Haskell Symposium*. https://doi.org/10.1145/2976002.2976018

Sam Lindley and J. Garrett Morris. 2016b. Embedding Session Types in Haskell. In *Haskell Symposium*. https://doi.org/10.1145/2976002.2976018

Sam Lindley and J. Garrett Morris. 2016c. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*.

Étienne Lozes and Jules Villard. 2011. Reliable Contracts for Unreliable Half-Duplex Communications. In *Web Services and Formal Methods - 8th International Workshop, WS-FM 2011, Clermont-Ferrand, France, September 1-2, 2011, Revised Selected Papers*. https://doi.org/10.1007/978-3-642-29834-9_2

Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. https://doi.org/10.4204/EPTCS.104.3

William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. https://doi.org/10.48550/arXiv.2207.06574

William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. OOPSLA (2017). https://doi.org/10.1145/3133911

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. In *HILT*. https://doi.org/10.1145/2663171.2663188

Jonathan May and Kevin Knight. 2006. Tiburon: A Weighted Tree Automata Toolkit. In *Implementation and Application of Automata*. https://doi.org/10.1007/11812128_11

Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. ICFP (2021). https://doi.org/10.1145/3473571

Stefan Milius, Dirk Pattinson, and Lutz Schröder. 2015. Generic Trace Semantics and Graded Monads. In *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015*. https://doi.org/10.4230/LIPIcs.CALCO.2015.253

Stefan Milius, Lutz Schröder, and Thorsten Wißmann. 2016. Regular Behaviours with Names. *Applied Categorical Structures* (2016). https://doi.org/10.1007/s10485-016-9457-8

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* (1978). https://doi.org/10.1016/0022-0000(78)90014-4

Robin Milner. 1980. *A Calculus of Communicating Systems*. https://doi.org/10.1007/3-540-10235-3

Fabrizio Montesi. 2021. Introduction to Choreographies. (2021).

Fabrizio Montesi and Marco Peressotti. 2018. Classical Transitions. *CoRR* (2018). arXiv:1803.01049 http://arxiv.org/abs/1803.01049

Edward F. Moore. 1956. *Gedanken-Experiments on Sequential Machines*. https://doi.org/doi:10.1515/9781400882618-006

Dimitris Mostrous and Nobuko Yoshida. 2015. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.* (2015). https://doi.org/10.1016/j.ic.2015.02.002

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP*. https://doi.org/10.1007/978-3-642-00590-9_23

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. https://doi.org/10.1007/3-540-45651-1

Magnus O. Myreen and Scott Owens. 2012. Proof-Producing Synthesis of ML from Higher-Order Logic. *SIGPLAN Not.* (2012). https://doi.org/10.1145/2398856.2364545

Hiroshi Nakano. 2000. A modality for recursion. In *LICS*. https://doi.org/10.1109/LICS.2000.855774

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations. OOPSLA (2019). https://doi.org/10.1145/3360587

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer. https://doi.org/10.1007/978-3-319-29604-3_5

Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2005. A Concurrent Lambda Calculus with Futures. https://doi.org/10.1007/11559306_14

Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2006. A concurrent lambda calculus with futures. *Theor. Comput. Sci.* (2006). https://doi.org/10.1016/j.tcs.2006.08.016

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR*. https://doi.org/10.1007/978-3-540-28644-8_4

Peter W. O'Hearn and David J. Pym. 1999. The Logic Of Bunched Implications. *Bulletin of Symbolic Logic* (1999). https://doi.org/10.2307/421090

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL*. https://doi.org/10.1007/3-540-44802-0_1

Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. https://doi.org/10.4204/EPTCS.211.7

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *CACM* (1976). https://doi.org/10.1145/360051.360224

Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *LICS*. https://doi.org/10.1145/2603088.2603116

Luca Padovani. 2017. A simple library implementation of binary sessions. *JFP* (2017). https://doi.org/10.1017/S0956796816000289

Brooks Paige, Frank Wood, Arnaud Doucet, and Yee Whye Teh. 2014. Asynchronous Anytime Sequential Monte Carlo. In *Advances in Neural Information Processing Systems 27*.

Robert Paige and Robert E. Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* (1987).

Rohit Parikh. 1985. The Logic of Games and its Applications. In *Topics in the Theory of Computation, Selected Papers of the International Conference on 'Foundations of Computation Theory', FCT '83*. `https://doi.org/10.1016/s0304-0208(08)73078-0`

Joachim Parrow. 1998. Trios in Concert. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*.

Marc Pauly. 2001. *Logic for Social Software*. Ph.D. Dissertation. `https://dare.uva.nl/search?identifier=9ad66ec5-063d-4673-8563-91369d0af7aa`

Arthur Paul Pedersen. 2014. Comparative Expectations. *Studia Logica* (2014).

David Peleg. 1987. Concurrent Dynamic Logic. *J. ACM* (1987). `https://doi.org/10.1145/23005.23008`

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *I&C* (2014). `https://doi.org/10.1016/j.ic.2014.08.001`

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FoSSaCS*. `https://doi.org/10.1007/978-3-662-46678-0_1`

Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). `https://doi.org/10.5555/509043`

Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Haskell Symposium*. `https://doi.org/10.1145/1411286.1411290`

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. ICFP (2021). `https://doi.org/10.1145/3473567`

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *PLDI*. `https://doi.org/10.1145/3385412.3386036`

Alexey Radul and Boris Alexeev. 2021. The Base Measure Problem and its Solution. In *AISTATS*. `http://proceedings.mlr.press/v130/radul21a.html`

Jason Reed. 2009a. *A Hybrid Logical Framework*. Ph. D. Dissertation. Carnegie Mellon University.

Jason Reed. 2009b. A Judgmental Deconstruction of Modal Logic. (2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. https://doi.org/10.1109/LICS.2002.1029817

Eric Roberts. 2001. An Overview of MiniJava. *SIGCSE Bull.* (2001). https://doi.org/10.1145/366413.364525

Pedro Rocha and Luís Caires. 2021. Propositions-as-types and shared state. ICFP (2021). https://doi.org/10.1145/3473584

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically Typed Compilation With Nameless Labels. POPL (2021). https://doi.org/10.1145/3434303

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *CPP*. https://doi.org/10.1145/3372885.3373818

Hannes Saffrich and Peter Thiemann. 2023. Polymorphic Typestate for Session Types. In *PPDP*. https://doi.org/10.1145/3610612.3610624

Alceste Scalas and Nobuko Yoshida. 2016a. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Alceste Scalas and Nobuko Yoshida. 2016b. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *POPL* (2019). https://doi.org/10.1145/3290343

Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. 2017. Nominal Automata with Name Binding. In *FoSSaCS 2017*. https://doi.org/10.1007/978-3-662-54458-7_8

Chung-Chieh Shan and Norman Ramsey. 2017. Exact Bayesian inference by symbolic disintegration, In POPL. https://doi.org/10.1145/3009837.3009852

Alexandra Silva, Filippo Bonchi, Marcello M. Bonsangue, and Jan J. M. M. Rutten. 2013. Generalizing determinization from automata to coalgebras. *LMCS* (2013). https://doi.org/10.2168/LMCS-9(1:9)2013

Alexandra Silva and Ana Sokolova. 2011. Sound and Complete Axiomatization of Trace Semantics for Probabilistic Systems. *Electronic Notes in Theoretical Computer Science* (2011). https://doi.org/10.1016/j.entcs.2011.09.027

Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *JFR* (2009). https://doi.org/10.6092/issn.1972-5787/1574

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI*. https://doi.org/10.1145/3453483.3454031

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. https://doi.org/10.1007/978-3-662-54434-1_32

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP*. https://doi.org/10.1007/978-3-642-54833-8_9

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_34

The Rust Team. 2023a. Fearless Concurrency. https://doc.rust-lang.org/book/ch16-00-concurrency.html

The Rust Team. 2023b. Reference Cycles Can Leak Memory. https://doc.rust-lang.org/book/ch15-06-reference-cycles.html

The Rust Team. 2023c. The Rust Programming Language. https://www.rust-lang.org/

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *PPDP*. https://doi.org/10.1145/3354166.3354184

Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. POPL (2020). https://doi.org/10.1145/3371135

Gavin Thomas. 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* (1997). https://doi.org/10.1006/inco.1996.2613

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank Wood. 2015. Output-Sensitive Adaptive Metropolis-Hastings for Probabilistic Programs. In *Machine Learning and Knowledge Discovery in Databases*. https://doi.org/10.1007/978-3-319-23525-7_19

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation*. Ph. D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *PPDP*. https://doi.org/10.1145/2003476.2003499

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_20

Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *FOSSACS*. https://doi.org/10.1007/978-3-319-89366-2_7

Bernardo Toninho and Nobuko Yoshida. 2019. Interconnectability of Session-Based Logical Processes. *POPL* (2019). https://doi.org/10.1145/3242173

Antti Valmari. 2009. Bisimilarity Minimization in $O(m \log n)$ Time. In *Applications and Theory of Petri Nets, PETRI NETS 2009*. https://doi.org/10.1007/978-3-642-02424-5

Antti Valmari. 2010. Simple Bisimilarity Minimization in O(m log n) Time. *Fundam. Informaticae* (2010). https://doi.org/10.3233/FI-2010-369

Antti Valmari and Giuliana Franceschinis. 2010. Simple $O(m \log n)$ Time Markov Chain Lumping. In *TACAS*.

Antti Valmari and Petri Lehtinen. 2008. Efficient Minimization of DFAs with Partial Transition. In *STACS*. https://doi.org/10.4230/LIPIcs.STACS.2008.1328

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. https://arxiv.org/abs/1809.10756

Rob J. van Glabbeek. 2001. The Linear Time - Branching Time Spectrum I. In *Handbook of Process Algebra*. https://doi.org/10.1016/b978-044482830-9/50019-9

Vasco T. Vasconcelos. 2012. Fundamentals of Session Types. *I&C* (2012). https://doi.org/10.1016/j.ic.2012.05.002

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *APLAS*. https://doi.org/10.1007/978-3-642-10672-9_15

John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. In *Monte Carlo Method*. Chapter 13.

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. https://doi.org/10.1145/2364527.2364568

Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction*.

Thorsten Wißmann, Hans-Peter Deifel, Stefan Milius, and Lutz Schröder. 2021. From generic partition refinement to weighted tree automata minimization. *Formal Aspects of Computing* (2021). https://doi.org/10.1007/s00165-020-00526-z

Thorsten Wißmann. 2023. Supported Sets – A New Foundation For Nominal Sets And Automata. In *Computer Science Logic (CSL'23)*. http://arxiv.org/abs/2201.09825

Thorsten Wißmann, Ulrich Dorsch, Stefan Milius, and Lutz Schröder. 2020. Efficient and Modular Coalgebraic Partition Refinement. *Logical Methods in Computer Science* (2020). https://doi.org/10.23638/LMCS-16(1:8)2020

Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *AISTATS*. http://jmlr.org/proceedings/papers/v33/wood14.html

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *I&C* (1994). https://doi.org/10.1006/inco.1994.1093

Yi Wu, Siddharth Srivastava, Nicholas Hay, Simon Du, and Stuart Russell. 2018. Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms. In *Proceedings of the 35th International Conference on Machine Learning*. http://proceedings.mlr.press/v80/wu18f.html

Uma Zalakain and Ornela Dardha. 2021. π with Leftovers: A Mechanisation in Agda. In *FORTE*. https://doi.org/10.1007/978-3-030-78089-0_9

Dan Zhang, Dragan Bosnacki, Mark van den Brand, Cornelis Huizing, Bart Jacobs, Ruurd Kuiper, and Anton Wijs. 2016. Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. In *MODELSWARD*. https://doi.org/10.1007/978-3-319-66302-9_13

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. OOPSLA (2020). https://doi.org/10.1145/3428216

Coq formalization index

Research data management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[1]

The following research code repositories have been produced during this Ph.D. research:

CHAPTER 1: Jules Jacobs, Stephanie Balzer, Robbert Krebbers; https://doi.org/10.5281/zenodo.5675249

CHAPTER 2: Jules Jacobs, Stephanie Balzer; https://doi.org/10.5281/zenodo.7150549

CHAPTER 3: Jules Jacobs, Stephanie Balzer, Robbert Krebbers; https://doi.org/10.5281/zenodo.6884760

CHAPTER 4: Jules Jacobs; https://doi.org/10.5281/zenodo.6560443

CHAPTER 5: Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers; https://doi.org/10.5281/zenodo.7993904

CHAPTER 6: Jules Jacobs, Jonas Kastberg Hinrichsen, Robbert Krebbers; https://doi.org/10.5281/zenodo.8422755

CHAPTER 7: Jules Jacobs; https://doi.org/10.5281/zenodo.4075076

CHAPTER 8: Jules Jacobs, Thorsten Wißmann; https://doi.org/10.5281/zenodo.7150706

Additionally, for the reader's convenience, the Coq mechanizations have been hosted on a separate web page that is interlinked with this thesis:

https://apndx.org/thesis/

---

[1] https://www.ru.nl/en/institute-for-computing-and-information-sciences/research, last accessed May 10, 2024

Summary

This thesis is about type systems for deadlock and leak free concurrency, separation logics for verified message passing, paradox-free probabilistic programming, and general and efficient coalgebraic automata minimization. In each case we aspire to guarantee beneficial properties 'by construction', as inherent consequences of the design of the system:

TYPES FOR DEADLOCK AND MEMORY LEAK-FREE CONCURRENCY:
Languages such as Rust guarantee memory safety and race freedom by type checking, but Rust programs can deadlock and leak memory. By enforing a carefully designed linear typing discipline, we obtain new concurrent languages with locks and channels in which deadlock freedom and memory leak freedom are guaranteed by type checking. A key challenge is proving this formally, for which we introduce a proof technique called *connectivity graphs*.

SEPARATION LOGICS FOR VERIFIED MESSAGE PASSING:
We show that concurrent separation logic and message passing are a perfect match, by using nested invariants to markedly simplify the soundness proof of an Actris-style separation logic for the verification of message passing programs. We then develop a new separation logic for verifying that message passing programs are deadlock free and memory leak free. These properties follow automatically from the linearity of the separation logic, without any additional proof obligations.

PARADOX-FREE PROBABILISTIC PROGRAMMING:
Probabilistic modeling languages suffer from paradoxes when conditioning on events of measure zero. This can even lead to different answers depending on whether the modeler is working in metric or imperial units. We show that we can avoid these paradoxes with a simple change to the modeling language. The resulting language guarantees that all probabilistic programs are invariant under change of parameterization, and its semantics of conditioning on events of measure zero relate to conditioning on events of positive measure in a natural way.

GENERAL AND EFFICIENT COALGEBRAIC AUTOMATA MINIMIZATION:
A variety of automaton types and minimization algorithms exist, often tailored to specific cases. By imposing a coalgebraic structure on automata, we can exploit their shared characteristics. This allows us to develop a unified minimization algorithm that works across a general class of automata, while being efficient both in theory and in practice.

Samenvatting

Dit proefschrift gaat over type systemen voor deadlock- en geheugenlek-vrije programmeertalen, separatielogica voor geverifieerde message-passing, paradox-vrije probabilistisch programmeren, en generieke en efficiënte coälgebraïsche automatenminimalisatie. In elk van de volgende gevallen streven we ernaar om gunstige eigenschappen inherent te garanderen:

TYPES VOOR DEADLOCK- EN GEHEUGENLEK-VRIJE CONCURRENCY:
Talen zoals Rust garanderen geheugenveiligheid en racevrijheid door middel van typechecking, maar Rust-programma's kunnen nog steeds vastlopen en geheugen lekken. Door een zorgvuldig ontworpen lineaire type-discipline verkrijgen we een taal met locks en channels waarin zowel deadlock-vrijheid als geheugenlek-vrijheid gegarandeerd zijn door typechecking. De belangrijkste uitdaging is dit formeel te bewijzen.

SEPARATIELOGICA VOOR GEVERIFIEERDE MESSAGE-PASSING:
We laten zien dat separatielogica en message-passing perfect samengaan, door gebruik te maken van geneste invarianten om de correctheid van een message-passing separatielogica aanzienlijk te vereenvoudigen. Vervolgens ontwikkelen we een nieuwe separatielogica voor het verifiëren dat message-passing programma's zowel deadlock-vrij als geheugenlek-vrij zijn. Deze eigenschappen volgen automatisch uit de lineariteit van de separatielogica, zonder enige extra bewijsverplichtingen.

PARADOX-VRIJ PROBABILISTISCH PROGRAMMEREN:
Probabilistische modelleertalen kampen met paradoxen wanneer ze conditioneren op gebeurtenissen van maat nul. Dit kan leiden tot uiteenlopende antwoorden die afhangen van de eenheden die de modelleur gebruikt (bijvoorbeeld metrische of imperiale lengtematen). We laten zien dat we deze paradoxen kunnen vermijden met een eenvoudige aanpassing van de modelleertaal. De resulterende taal garandeert dat alle probabilistische programma's invariant zijn onder verandering van parameterisatie, en de semantiek van conditionering op gebeurtenissen van maat nul op een natuurlijke manier gerelateerd is aan conditionering op gebeurtenissen van positieve maat.

GENERIEKE EN EFFICIËNTE COÄLGEBRAÏSCHE AUTOMATENMINIMALISATIE:
Er bestaan verschillende automaatminimalisatiealgoritmen, toegespitst op specifieke gevallen. Door een coälgebraïsche structuur op automaten te leggen ontwikkelen we een generiek minimalisatiealgoritme dat zowel theoretisch als in de praktijk efficiënt is voor een algemene klasse van automaten.

Curriculum vitae

Jules Jacobs was born on October 17, 1989 in Nijmegen, the Netherlands. He studied mathematics and physics (bachelor), and mathematics (master) at Leiden University, obtaining a cum laude grade for his master thesis on the topic of differential topology.

He started his PhD research at Delft University of Technology, under the supervision of Robbert Krebbers and Eelco Visser, later continued at Radboud University Nijmegen, under the supervision of Robbert Krebbers and Herman Geuvers, with Stephanie Balzer from Carnegie Mellon University as copromotor. His PhD research focused on mathematical foundations of programming languages, in particular type systems for deadlock free concurrency, separation logic, coalgebraic minimization, and probabilistic programming. During his PhD, he also helped teach courses in this area, and co-supervised bachelor and master students. Jules is currently a postdoc at Cornell University, working with Nate Foster and Alexandra Silva.

Titles in the IPA Dissertation Series since 2021

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of

Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction*. Faculty of Science, Mathematics and Computer Science, RU. 2024-07