

Strong Program Logics for Weak Memory and Even Stronger Types for Tactic Programming

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von Jan-Oliver Kaiser

Saarbrücken, 2025

TAG DES KOLLOQUIUMS
19. März, 2026

DEKAN DER FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
Prof. Dr. Roland Speicher

PRÜFUNGSAUSSCHUSS

Vorsitzender: Prof. Dr. Sebastian Hack

Gutachter: Prof. Dr. Derek Dreyer

Prof. Dr. Deepak Garg

Dr. Edwin Brady

Akademischer Mitarbeiter: Dr. Travis Hance

Abstract

Computers have become ubiquitous in everyday life and so have bugs in programs running on those computers. By the author's own estimate, not a single piece of software actually works correctly. More scientific estimates are hard to come by. Nonetheless, there is no shortage of high-profile security incidents due to software defects. Research in the field of programming languages and verification has produced countless ways to attack the problem of software defects. This thesis concerns itself with two techniques related to verification, both well established in their original application and, here, being applied to an unconventional setting.

Firstly, in the category of extending the applicability of program verification to more realistic settings, we demonstrate how to verify high-performance algorithms and data structures making use of highly efficient memory access patterns that are not automatically synchronized with main memory or other processors' caches. Hardware and programming languages that expose these weakly or un-synchronized memory accesses are said to have *weak memory models*. The lack of synchronization in weak memory models goes directly against the fundamental assumption of sequential consistency, *i.e.*, the assumption that all memory accesses are fully synchronized, which still sits at the heart of most verification works. Concretely, we show how to perform verification in weak memory models using an existing program logic framework, *Iris*, that was traditionally limited to sequential consistency. In building on *Iris*, we inherit its mechanized proof of soundness of the core logic as well as the ability to perform mechanized program verification.

Secondly, in the category of verification infrastructure, we propose the concept of typed tactics which bring the benefits of dependent types to the process of writing and automating proofs in the Rocq proof assistant. This endeavor aims to support verification efforts whose scale is quickly approaching the limits imposed by Rocq's existing support for proof automation, *i.e.*, its built-in tactic language called *Ltac*. *Ltac*'s limitations and pitfalls are numerous and many of them can be traced back to design decisions made over two decades ago when the language was created. Since then, Rocq itself as well as how it is being used has changed and *Ltac* is no longer able to fulfill the requirements of large verification projects. Our contribution is a new tactic language called *Mtac2*. *Mtac2* is based on *Mtac*, a principled metaprogramming language for Rocq offering strongly typed primitives based on Rocq's own dependent type system. *Mtac*'s primitives could already be used to implement some tactics but the language lacks the ability to directly interact with Rocq's proof state and perform backwards reasoning on it. *Mtac2* extends *Mtac* with support for backwards reasoning and keeps in line with *Mtac*'s tradition of strong types by introducing the concept of typed tactics. Typed tactics statically track the expected type of the current goal(s) and can rule out entire classes of mistakes that are often found in *Ltac* code.

Zusammenfassung

Computer sind ein nicht wegzudenkender Teil unseren alltäglichen Lebens; genau so wie Fehler in den Programmen, die auf diesen Computern ausgeführt werden. Schätzung des Authors zufolge, gibt es kein einziges Programm, dass tatsächlich korrekt funktioniert. Objektivere Schätzungen hierzu sind nicht leicht zu finden. Anders hält es sich aber mit aufmerksamkeitswirksamen Berichten über Sicherheitslücken auf Grund von Software-Fehlern, denn die findet man schnell und immer öfter. Die Forschung im Bereich der Programmiersprachen und Verifikation hat vielerlei Lösungsansätze hervorgebracht, die mit dem Problem der Software-Fehler helfen sollen. Diese Thesis beschäftigt sich mit zwei solcher Techniken, beide wohl etabliert in ihrem ursprünglichen Anwendungsgebiet, und wendet sie auf unkonventionelle Art und Weise an.

Erstens zeigen wir, wie eine existierende Technik zur Programm-Verifikation ausgeweitet werden kann, um damit auch solch nebenläufige Algorithmen korrekt beweisen zu können, die sich—aus Effizienzgründen—nicht auf automatische Synchronisierung zwischen Prozessorkernen untereinander oder mit dem Hauptspeicher verlassen. Bei Hardware und auch Programmiersprachen, die diese schwach oder gar nicht synchronisierten Speicherzugriffsmuster zulassen, spricht man von sogenannten *schwache Speicherkonsistenzmodellen* (engl. *weak memory models*). Die fehlende automatische Synchronisierung widerspricht einer Grundannahme, die immer noch den Kern fast aller Verifikationsforschung zugrunde liegt: die sequentielle Konsistenz, also die Annahme, dass alle Speicherzugriffe vollständig synchronisiert sind. Wir zeigen, wie die Verifikations-Logik *Iris*, die ursprünglich auf Verifikation-Szenarien mit sequentieller Konsistenz beschränkt war, zur Verifikation unter schwachen Speicherkonsistenzmodellen genutzt werden kann. Dadurch, dass wir auf *Iris* aufbauen, erhalten wir ohne weiteren Zusatzaufwand sowohl einen Korrektheitsbeweis der sogenannten Kernlogik als auch die Fähigkeit, Computer-geprüfte Korrektheitsbeweise von Programmen durchzuführen.

Zweitens schlagen wir im Bereich der Verifikations-Infrastruktur das Konzept von getypten Beweis-Taktiken vor, die den Prozess des Verfassens und vor allem das Automatisierens von Beweisen im Beweisassistenten Rocq mit Hilfe von *dependent types* vereinfachen. Diese Bestrebung soll denjenigen Verifikationsprojekten dienen, die aus Rocqs eigenen Werkzeugen zur Beweisautomatisierung herausgewachsen sind. Vor allem geht es hierbei um Rocqs eingebaute Taktiksprache *Ltac*. *Ltac* ist eine sehr beschränkte Taktiksprache und viele der immer noch bestehenden Fehler in ihrem Design lassen sich bis zu der Einführung der ersten Version vor über 20 Jahren zurückverfolgen. Rocq selber hat sich allerdings seitdem stark verändert und die Anforderungen der Nutzer ebenso. Kurz gesagt: *Ltac* erfüllt nicht die Voraussetzungen für großangelegte Verifikationsunternehmungen. Unser Beitrag ist eine neue Taktiksprache, *Mtac2*. *Mtac2* ist der Nachfolger von *Mtac*, einer prinzipienstarken Metaprogrammiersprache für Rocq, die Rocqs eigenes Typsystem benutzt, um stark getypte Programmierbausteine bereitzustellen. *Mtac* selbst konnte schon eingesetzt werden, um manch einfache Taktiken zu implementieren, aber es fehlte sowohl die Möglichkeit, direkt den Zustand der Beweismachine in Rocq zu manipulieren, als auch die Möglichkeit des reduktiven Rückschlusses (engl. *backward reasoning*). *Mtac2* schließt diese Lücken indem es *Mtac* um getypte Taktiken erweitert. Getypte Taktiken folgen zu jeder Zeit statisch dem Typ des Beweisziels und können dadurch ganze Klassen von Fehlern unmöglich machen, die sonst oft *Ltac*-Programme plagten.

Acknowledgements

First and foremost, I am deeply grateful to my advisor, Derek Dreyer. There is not much I can say that has not already been said about Derek's technical expertise and his ability to turn half-baked sentences into academic gold. I can only echo the many sentiments of my PhD siblings. If that was all he was good at, it would have been enough for me to be incredibly privileged to work with him. But Derek is also a good person, somebody that I could always trust and who always had my back. Derek has been incredibly supportive and I truly cannot imagine that I could have come this far without that support. For many years, his belief in my abilities and my potential far surpassed my own. I am truly thankful to him for always seeing the best version of me during the many years we have known each other. I came to Derek looking for an advisor and I am delighted to have also found a mentor and a friend.

Many, many thanks go to a long list of friends and colleagues that I had the privilege of working with during my time at MPI-SWS. Beta Ziliani deserves special mention for getting me started when I began my PhD, for his continued support and encouragement even after he finished his own PhD and moved to a different country, and just generally for being a great friend. Without him, this thesis would not have been written. I am very grateful to my friend and PhD sibling Ralf Jung for mentioning MPI-SWS and Derek to me when I was lost in grad school and for traveling around the world with me. Also deserving of special mention is my friend, PhD sibling, and collaborator Hoang-Hai Dang who was instrumental in the success of the iGPS project. I fondly remember the countless hours we spent in front of whiteboards during which he patiently entertained every hare-brained scheme I would come up with. I also owe many thanks to my friend and colleague David Swasey for lending an empathetic and helpful ear whenever I needed one.

Edwin Brady and Deepak Garg deserve special thanks for reviewing my thesis, their feedback, and for their questions and suggestions during the defense.

For encouraging me to apply to Saarbrücken Graduate School of Computer Science, I am thankful to Professor Smolka. I also have him to thank for my first exposure to the field of computational logic and the proof assistant Rocq which became the foundation of my work both in- and outside of academia.

I am deeply indebted to the original and the current team behind the Iris project for making Iris so successful that it has made its way into industry where I am being paid to use it. While my joining industry may look like it delayed this thesis I am convinced that it also gave me a much needed change of perspective without which I might have never finished my PhD.

Rose Hoberman deserves special mention for her classes on academic writing and presentations that teach not only what not to do, but what to do instead. (If this thesis is hard to read it is not because of the tools that were available to me but because of how I chose to use them.) She is also a great friend who always has useful advice, usually shared over delicious home-cooked food.

Special thanks also go to the MPI-SWS administration who were incredibly helpful during the entire time I was with the institute: Claudia Richter, Annika Meiser, and Maria-Louise Albrecht. I also had the immense privilege of being supported by the most helpful IT staff I have ever worked with, Christian Klein and Carina Schmitt.

Finally, I want to thank Andrea Fischer for supporting me at every step of this journey, for encouraging me to seek help when I needed it, for not ever giving up despite my stubbornness, and for being a wonderful partner.

Contents

PROLOGUE	xiii
1 OVERVIEW	1
1.1 iGPS	1
1.2 Mtac2	6
1.3 Structure, Publications, and Collaborations	9
BIBLIOGRAPHY	11
 I STRONG LOGIC FOR WEAK MEMORY: REASONING ABOUT RELEASE-ACQUIRE CONSISTENCY IN IRIS	 15
2 INTRODUCTION	19
2.1 Iris: A Unifying Framework for Concurrent Separation Logics . . .	19
2.2 Separation Logics for Release-Acquire Consistency	20
2.3 Our Contributions	22
3 RELEASE-ACQUIRE AND NON-ATOMICS	25
3.1 Release-Acquire	26
3.2 Non-Atomics	27
3.3 The λ_{RN} language	29
4 IRIS	31
4.1 Iris by Example	32
4.2 Instantiating Iris with λ_{RN}	37
5 IGPS	41
5.1 Key Features of GPS	41
5.2 Single-Writer Protocols	45
5.3 The Model of iGPS	47
6 OTHER CONTRIBUTIONS	49
7 RELATED WORK	53
APPENDICES	55
APPENDIX A OPERATIONAL SEMANTICS FOR RA+NA	57
A.1 Declarative Semantics	57
A.2 First operational semantics	63
A.3 Timestamp machine	67
APPENDIX B SINGLE-WRITER PROTOCOL PROOF RULES	71

B.1	Escrow and exchange proof rules	71
B.2	RSL proof rules	73
B.3	Bounded ticket locks	75
B.4	Proof setup for bounded ticket locks	76
B.5	Proof steps for lock	79
APPENDIX C IGPS PROTOCOL MODEL		83
BIBLIOGRAPHY		87
II MTAC2: TYPED TACTICS FOR BACKWARD REASONING IN ROCQ		93
8	INTRODUCTION	97
8.1	An example motivating typed backward reasoning	98
8.2	Contributions	102
9	THE CORE OF MTAC BY EXAMPLE	103
10	MTAC2: ADDING BACKWARD REASONING TO MTAC1	109
10.1	The tactic type	111
10.2	The goal type	112
10.3	Tactic composition	116
10.4	Returning values from tactics	119
10.5	Examples	120
10.6	Concluding remarks	124
11	TYPED BACKWARD REASONING	125
12	CASE STUDY: IRIS PROOF MODE	131
12.1	Symbolic Execution in Ltac	132
12.2	Problems with the Ltac Implementation	134
12.3	Symbolic Execution in Mtac2	135
12.4	Benefits of the Mtac2 Implementation	138
12.5	Symbolic execution of heap operations	139
13	OPERATIONAL SEMANTICS OF MTAC2	145
13.1	Primitives and Their Types	146
13.2	Introducing Mtac2's Virtual Machine	149
13.3	Typing the Virtual Machine	150
13.4	Semantics of the Virtual Machine	153
13.5	Theory	167
14	TECHNICAL DETAILS CONCERNING SOME NEW FEATURES OF MTAC2	177
14.1	Polymorphic universes	177
14.2	Sort polymorphism	178
14.3	Opaque definitions	179
15	RELATED AND FUTURE WORK	181
15.1	Future work	185
BIBLIOGRAPHY		187

III MTAC2: ENGINEERING CHALLENGES	195
16 THE INTERPRETER	199
16.1 Reduction Machine	200
17 UNIVERSES	203
17.1 Background: Universes in Rocq	203
17.2 Universes in Mtac2	208
17.3 Performance Issues with Universe Polymorphism	208
17.4 Universe-Polymorphic Recursion	210
18 FURTHER CHALLENGES	219
18.1 Meta- and Tactic Programming: Not Entirely the Same	221
18.2 Modeling Rocq in Rocq, Partially	223
18.3 Missing Support for Unsafe Operations	230
BIBLIOGRAPHY	233

Prologue

1

Overview

The following report describes two results in seemingly unrelated corners of verification and programming languages research:

1. iGPS: A fully mechanized program logic for the Release-Acquire fragment of the C11 weak memory model
2. Mtac2: A very strongly typed tactic language for the Rocq proof assistant¹

¹ The Rocq proof assistant was previously known as Coq and has been renamed with its 9.0 release in March, 2025.

In this overview I will introduce these two projects, outline the context in which they came into existence, and explain how they are in fact not as unconnected as they may appear.

1.1 iGPS: A FULLY MECHANIZED PROGRAM LOGIC FOR THE RELEASE-ACQUIRE FRAGMENT OF THE C11 WEAK MEMORY MODEL

To understand the goal of the iGPS project we need to first understand the concept of *memory models*. Memory models are used to assign meaning to a program's interaction with the main memory, which in this context is often referred to as the *heap*. Broadly speaking, memory models specify the behavior of reads, writes, and updates² to the heap. These specifications can then be relied upon by programmers to reason about the behavior of their program. For example, in a single-threaded program one would expect that after writing a value v to a location x in memory, reading from x will yield v .

We would write such a program the following way. The variable a denotes a local register that is distinct from variables in memory.

```
x := v;  
a := x; // must read v
```

² Updates are operations that perform reads and then, possibly, depending on the value they read, writes. CAS (compare and swap) and FAI (fetch and increment) are prominent examples.

For us, the most interesting part of a memory model is its specification of programs in which multiple threads interact with memory concurrently. In a multi-threaded setting, one thread could first write value v to x and then write another value w to x . Concurrently, a second thread could read

twice from x . This program can be written the following way, in which we initialize x to some initial value i .

```
x := i;
x := v; || a = x; // possible values?
x := w; || b = x; // possible values?
```

A memory model specifies the allowed outcomes of the two read operations. Given that we make no assumptions about how the two threads are scheduled, we could expect a memory model to allow for both reads to return the initial value i . The same is likely true for reading (i, v) , (v, v) , (v, w) , and (w, w) .

[Lam79] Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”

The remaining outcomes are where *weak* memory models get their distinction. We say that a memory model is *sequentially consistent* [Lam79] (or implements *sequential consistency*) if the following is true: If the second thread reads any combination of v and w , it has to be (v, w) , *not* (w, v) . More generally, sequential consistency demands that all observations are consistent with a sequential interleaving of the threads’ execution.

Conversely, memory models that do allow for the outcome (w, v) are not sequentially consistent and thus offer *weaker* guarantees (*i.e.*, allow *more* behaviors). Those models are called *weak memory models*.

The specifications provided by memory models translate to sets of allowed optimizations (*e.g.*, reorderings of instructions, elimination of instructions) that compilers and even the hardware running the program may be allowed to perform without violating the model’s guarantees. In our single-threaded write-then-read example, an optimizing compiler might eliminate the read operation entirely since the return value is guaranteed to be v . Similarly, a sufficiently smart processor could decide to never flush the written value to main memory (*i.e.*, never make it visible to other cores) if the program is only executed on a single core.

The question of which optimizations are allowed at the hardware level is most pressing in the setting of multi-core processors that come equipped with local caches. When multiple processors operate on overlapping parts of the heap, they generally need to perform expensive steps to keep their local caches synchronized with each other. Hardware engineers have developed different cache synchronization strategies to mitigate this cost as much as possible. However, the general trend of growing caches and adding more, bigger cache layers works to counteract the improvements gained through smarter strategies. It is still generally true that the cost of synchronization far outweighs that of interacting solely with local caches.

Modern processor architectures thus opt for compromises on the performance-synchronization trade-off and leave it up to compilers (and developers writing performance critical code) to explicitly insert the necessary synchronization primitives when the hardware’s built-in synchronization is not enough. These architectures come equipped with their own memory model that allows developers to understand which primitives need to be inserted to achieve the desired level of synchronization.

1.1.1 C11 Memory Model

Most programs, even performance critical programs, are not written exclusively for a single processor. Similarly, most programs are not written in any specific assembly dialect that would grant direct access to the processor's primitive synchronization operations. Instead, programs are written in high-level languages and it is the job of a compiler to translate the code into an executable that is suitable for the target processor.

To be able to determine that the compilation result is correct, source languages need their own memory models with their own set of guarantees which the compiler must preserve during compilation. One prominent language that offers a detailed memory model is the C programming language. The model was introduced in a revision of the C standard called ISO/IEC 9899:2011 which is generally referred to as *C11*. Its memory model is thus referred to as the *C11 memory model*.

The C11 memory model is not only the de-facto standard for most performance critical work, it also serves as a template for other programming languages [Dan+20]. This effect is likely further enhanced due to the fact that C is the main source language for the two biggest compilers, gcc and llvm.

This omnipresence arguably stands in stark contrast to the complexity of the model. Programmers already struggle with undefined behavior and even mature projects suffer from miscompilations [Wan+13] because of it. The C11 memory model introduces additional opportunities for undefined behavior for developers aiming to maximize performance in multi-threaded programs, *i.e.*, in code that is usually already very complex.

Defining the C11 memory model in detail would eclipse the scope of this introduction. Instead, we give a short overview of the main way that developers interact with the model when writing concurrent code: choosing consistency levels for heap operations. Reads and writes must choose one level whereas updates (*i.e.*, reads that are conditionally combined with writes, *e.g.*, compare-and-swap), can generally choose different levels for the read and write part.

Non-Atomic Accesses

The weakest level of consistency is offered by non-atomic (NA) accesses. Non-atomic accesses are generally compiled without any synchronization primitives. All operations in C are non-atomic unless the developer indicates otherwise.

Non-atomic accesses are forbidden from *racing* with other non-atomic and even atomic accesses on the same heap location. The exact definition of a race is rather involved. Intuitively, a race can only happen when two accesses are not synchronized by operations using the stronger consistency levels described below. Such a race involving non-atomic accesses, called a *data race*, is *undefined behavior* in C11. Undefined behavior is a technical term used by the C standard to designate situations which compiler developers can assume to never happen.

That assumption allows compilers to perform optimizations on non-

[Dan+20] Dang et al., “RustBelt meets relaxed memory”

[Wan+13] Wang et al., “Towards optimization-safe systems: analyzing the impact of undefined behavior”

atomic operations that one would generally expect in single-threaded code. For example, a non-atomic read preceded immediately by a non-atomic write to the same location can be elided since the value is entirely determined by that preceding write. This optimization is not behavior preserving in the presence of multiple threads writing to that same location concurrently since the value could then have changed between the write and read operations. C11 opts to put the onus of ensuring that this situation does not arise on the developer, allowing the compiler to optimistically perform such optimizations.

Relaxed

Relaxed (*RLX*) is the first class of atomic accesses offered by C11. Relaxed accesses are usually implemented without any synchronization primitives at the hardware level. However, in contrast to non-atomic accesses, they are allowed to race with other atomic accesses to the same heap location. The compiler is *not allowed* to perform the read elision optimization described above. But it may well perform other optimizations.

Since relaxed accesses do not synchronize, they cannot be used to implement synchronizing abstractions such as atomic flags and locks.

Release-Acquire

Release-acquire (*RA*) accesses are the weakest synchronizing atomic accesses in C11. As the name implies, there release-acquire consists of *releasing* and an *acquiring* action. Writes *release* changes to other threads which can then perform reads to *acquire* those changes. This is best illustrated by the canonical message passing example of *release-acquire*.

```

x := 0;
y :=_na 42 || while (x_acq == 0);
x :=_rel 1 || a := y_na    // must read 42

```

The code shows two threads cooperating to safely transfer a message contained in *y* from the left-hand-side thread to the right-hand-side thread. The LHS thread writes the message non-atomically, *i.e.*, in an unsynchronized way, to *y* and then changes the value of the flag variable *x* using a *release* write. Meanwhile, the RHS thread waits for the flag *x* to become non-zero. As soon as it does, it reads the message from *y* into a local variable *a*. The use of release-acquire accesses guarantees that the non-atomic variable *y* can be written and then read without causing a data race and, also, that the result of the read is the message value.

Section 3 goes into more detail about *release-acquire* (and non-atomic). For now, it suffices to understand that release-acquire accesses can be used to establish synchronization.

Sequential Consistency

Sequentially consistent (*SC*) accesses are the strongest class of synchronizing accesses. As the name suggests, SC aims to provide accesses that

behave consistently with a sequential execution. This intuitive understanding is complicated by the fact that SC accesses can co-exist with other atomic and non-atomic accesses. Defining the interactions of these different accesses has been difficult and the initial attempts contained bugs[Lah+17]. Sequentially consistent accesses are not relevant to iGPS and will thus not be discussed further.

It is hopefully clear from the different levels of consistency offered by C11 that weak memory in general and this model in particular are significantly more complex than sequentially consistent models. While some argue that the added complexity does not pay off [Mar+15; LMM17], it seems the C11 memory model and its weaker consistency levels are here to stay. With the added complexity come important challenges for programming languages and verification researchers. Chief among them is the fact that decades of existing research has focused almost exclusively on sequential consistency and there is no general recipe to retrofit an existing verification approach or tool with a weaker memory model.

1.1.2 Modular Verification

Consequently, there has been a plethora of research related to weak memory models in the PLV community in the last decade. The most fundamental work related to the C11 memory model arguably comes in the form of a fully formal mathematical model[Bat+11]. The formalization does fully explain the behavior of weak memory operations. However, the way it does so makes it *inconvenient* to use for modular program verification. Due to the time-traveling way that undefined behavior can impact earlier steps in the execution, C semantics generally cannot be expressed in a small-step way. The weak memory aspect of C11 is no different. Its formalization similarly characterizes the validity of entire executions, not individual steps.

In the context of program logics, *i.e.*, special-purpose logics built to verify the correctness of program code, a lot of work can be saved by building on small-step semantics. Being able to locally characterize the effect of, for example, heap operations on the global state greatly simplifies the correctness proof of the program logic. Of particular note for the iGPS project is the ability to plug small-step semantics into the logic framework *Iris* which then provides building blocks to derive a full program logic for the provided semantics.

To unlock these advantages, iGPS relies on a restricted fragment of Release-Acquire, called Strong Release Acquire[LGV16] (SRA). SRA provides a small-step, localized semantics to the Release-Acquire fragment of the C11 memory model. This semantics is fundamental to iGPS and its use of *Iris* to construct a program logic for Release-Acquire. That logic is an extended version of *GPS*[TVD14], short for Ghosts, Protocols, and Separation. *GPS* was the first program logic to offer modern program logic features such as ghost state and (single-location) protocols under weak memory. In building on the *Iris* framework, iGPS manages to derive these features on top of existing, generic constructions offered by the framework. iGPS is fully mechanized in the Rocq proof assistant and

[Lah+17] Lahav et al., “Repairing sequential consistency in C/C++11”

[Mar+15] Marino et al., “The Silently Shifting Semicolon”

[LMM17] Liu et al., “A volatile-by-default JVM for server applications”

[Bat+11] Batty et al., “Mathematizing C++ concurrency”

[LGV16] Lahav et al., “Taming Release-acquire Consistency”

[TVD14] Turon et al., “GPS: navigating weak memory with ghosts, protocols, and separation”

[KTB17] Krebbers et al., “Interactive proofs in higher-order concurrent separation logic”

[Kre+18] Krebbers et al., “MoSel: a general, extensible modal framework for interactive proofs in separation logic”

[CP88] Coquand et al., “Inductively defined types”

[PP89] Pfenning et al., “Inductively Defined Types in the Calculus of Constructions”

[TS18] Timany et al., “Cumulative Inductive Types In Coq”

³ It seems likely that the term “tactic language” was chosen because proof scripts contain the *tactic*, or *strategy*, of the proof. The author could not find any confirmation of this.

inherits Iris’s ability to perform verification of programs entirely in Rocq thanks to Iris Proof Mode[KTB17; Kre+18]

1.2 MTAC2: A VERY STRONGLY TYPED TACTIC LANGUAGE FOR THE ROCQ PROOF ASSISTANT

The work on a tactic language for Rocq is not directly related to weak memory or program logics. However, it is motivated in part by the experience of formalizing iGPS. The formalization of iGPS involved adapting existing programming-language specific proof automation within the Iris project to the weak memory setting. Conceptually, these changes were trivial and largely followed the changes in the actual definitions. However, as is often the case in the author’s personal experience, the changes to the automation required an outsized amount of debugging and were cause for an ever greater amount of frustration.

To properly convey this frustration, I will give a brief overview of how the Rocq proof assistant works. On the most basic level, Rocq is a typechecker that checks the types of programs written in a heavily extended variant of the *Calculus of Inductive Constructions*[CP88; PP89; TS18]. This typechecker within Rocq is called the *kernel*. Due to the Curry-Howard correspondence, the act of typechecking corresponds to proof checking and it is this ability to check proofs that Rocq is most known for. A consequence of this design is that Rocq proof objects are *proof terms*: programs whose type is exactly the proposition that is meant to be proven.

Rocq proofs are not generally written directly as proof terms. There are many reasons for this: proof terms are hard to write, even harder to read, almost impossible to maintain, and ultimately largely irrelevant to the mathematical idea at the core of a proof. Rocq thus offers the ability to interactively write proofs using *proof scripts*.

Proof scripts consist of a sequence of sentences, each potentially executing many primitive steps to manipulate the *goals* of the proof. The initial goal is the theorem statement. Proof steps—such as introducing quantifiers, performing case analysis, or applying other theorems—may open new goals (often called *sub-goals*). The intermittent goals can be inspected interactively, providing live feedback during the construction (and maintenance) of proofs.

The language in which proof scripts are expressed is usually called a *tactic language*.³ The individual proof steps of tactic languages are often called *tactics*. These usually include primitive operations such as applying theorems to the current goal, instantiating an existential in the conclusion with a given witness, or—if the language permits this—an application of a custom tactic. Utterances that combine and modify tactics are sometimes called *tacticals*. These often include logical combinators (try tactic X, if it fails try Y), selecting subsets of goals to which to apply tactics, etc.

When all goals of the proof have been closed by sentences in the proof script and the proof is finished (usually by the iconic *Qed* sentence), the

resulting proof term is sent to Rocq kernel to be typechecked. It may seem unnecessary to re-check a proof whose individual steps were already accepted by Rocq. However, this step is necessary because proof scripts are not part of Rocq trusted computing base which, conceptually, contains only Rocq kernel. In fact, proof scripts are entirely untrusted and may produce ill-typed proof terms and bogus goals. The separation between proof scripts and the final typechecking step is known as the De Bruijn criterion[BW05] (sometimes “De Bruijn principle”). This design is the reason why Rocq can be agnostic to the choice of tactic language.

Despite this flexibility, Rocq has for the longest time⁴ shipped exclusively with a single tactic language called *Ltac*[Del00]. The rest of this thesis will paint *Ltac* in an arguably realistic but ultimately deeply unfavorable light. To balance this critique, it is worth noting that *Ltac* started as a special-purpose language and—not least due to the strong commitment to backwards compatibility—could never be fully re-designed to rectify design decisions that have since turned out to be counterproductive. Still, the language has evolved since its inception in order to (attempt to) fulfill the needs of ever-growing formalization projects that were undertaken in Rocq and many, maybe most, success stories of formalizations benefited from *Ltac*. To a large degree, Rocq owes its success to *Ltac*.⁵

At the same time, it is no secret that *Ltac* is perceived to be holding back Rocq. Formalization projects keep outgrowing the language’s features, suffer from its many performance issues, and stumble over the seemingly impossible task of writing maintainable proof automation in a language that is made up of hundreds of incongruous tactics using different conventions for syntax, names, input terms, reduction strategies, unification strategies, error messages, etc. It is thus fitting that *Ltac* has been compared to PHP[Péd16]; another language that was borne out of a very special purpose need and that grew to keep up with requested features to eventually become the poster child of bad programming languages *while simultaneously being one of the most successful programming languages in its domain*.

To put the critique of *Ltac* into more concrete terms let us briefly consider some of the language’s properties and problems:

- *Ltac* tactics are statically untyped.
- *Ltac* does not provide access to the AST of Rocq terms.
- *Ltac* has no notion of user-defined types (and only a few built-in types exist).
- *Ltac* offers many powerful built-in tactics but not nearly enough useful primitives for users to define similarly powerful custom tactics.
- *Ltac* is backtracking-first, *i.e.*, almost all language constructs will interpret failure as a request to find another way to proceed.⁶
- *Ltac* backtracking cannot be done modularly: Failure is always handled by the innermost backtracking point except when users specify a *fixed offset* in the form of an integer which then skips as many handlers.

[BW05] Barendregt et al., “The challenge of computer mathematics”

⁴ Until version 8.11, released in November 2019, when *Ltac2* was officially added to Rocq.

[Del00] Delahaye, “A Tactic Language for the System Coq”

⁵ That is not to say that no other tactic language could have done a better job; it is only meant to say that no other tactic language was available to do *Ltac*’s job.

[Péd16] Pédrot, “Towards *Ltac* 2.0”

⁶ Some constructs have a non-backtracking variant but users need to remember to use it. E.g. the `lazymatch` variant of the term introspection tactic `match .. with .. end`.

- Ltac favors unhygienic tactics by making them the default and requiring extra work for hygiene. (Ltac `foo := intros x` versus Ltac `foo := let x := fresh "x" in intros x.`)
- Ltac performs binding analysis but hard-codes the set of binding-introducing tactics. (Ltac `foo := intros x; destruct x` is allowed but Ltac `foo := custom_intros x; destruct x` is not.)

The list could be extended far beyond its current items and Section 12.2 goes into more detail on some of the aspects listed above. Setting aside the question of completeness, the list should suffice to understand how frustrating working with Ltac can be. The frustration is only increased by the fact that Rocq term and specification language, Gallina, is strongly, even dependently typed, and fully extensible due to its support for custom (inductive) types. And while Gallina is not known as a particular great general-purpose programming language the fact that it is in many ways more convenient for that task than Ltac, a tactic language that suffers from almost none of the design constraints inherent to Gallina is a particularly glaring discrepancy.

[Zil+13] Ziliani et al., “Mtac: a monad for typed tactic programming in Coq”

⁷ With an interpreter implemented as a Rocq plugin.

It is thus perhaps no surprise that people eventually sought to bring the advantages of Gallina to the world of tactics: The Mtac language[Zil+13] offers a monadic metaprogramming language that is expressed entirely in Gallina.⁷ Metaprograms in Mtac are Gallina terms that have Gallina types and they benefit from Rocq dependent type system. Most importantly, the type of Mtac programs is indexed by the expected type of the terms generated by the metaprogram. Mtac programs are guaranteed to either loop, fail, or produce a value of exactly that expected type.

Mtac came equipped with primitives that were sufficient for a variety of metaprogramming tasks but it fell short of being a full-fledged tactic language for several reasons (see Section 8). Mtac2, the successor of Mtac, seeks to improve on Mtac in offering a tactic language that marries the advantages of Mtac’s typed metaprogramming with the flexibility that is often required in writing widely applicable tactics. Mtac2 provides a new monad of typed tactics that inherits the advantages of Mtac’s original monad and also allows easy integration of *dynamically-typed tactics*.

The work on Mtac2 represents the heart of this thesis. Part II contains an updated and much improved version of the published Mtac2 paper. The most important change is the addition of a proof of type soundness of the core language.

The work on typed tactics represents the single biggest individual Mtac2-related project during the author’s PhD—and the only published one—but it is only one part of the overall work and the project’s bigger vision. A substantial amount of time was invested in attempting to evolve Mtac2 from a research prototype into a production-ready tactic language. A lot of time and energy of that endeavor was put into optimizing Mtac2’s performance.

1.2.1 The Quest for Performance

While the published results on Mtac2 argue in depth for the language’s usefulness in the tactic development process, they only mention the aspect of runtime performance in passing. At the time these results were published, microbenchmarks already favored Mtac2 over Ltac in several examples. Unfortunately, more realistic case studies taken from the Iris project were significantly slower than their Ltac counterparts. These examples in particular informed the following work on improving the performance of Mtac2.

There is good reason, in theory, to expect stellar performance from Mtac2: its strong type guarantees should allow for a whole host of optimizations in the interpreter. The flip side of that coin is that some things necessarily need to be checked dynamically in order to uphold these strong guarantees. In practice, however, the biggest obstacle in Mtac2 performance is Mtac2’s strong ties to its host system, Rocq. There are several key aspects to the interaction of Mtac2 with Rocq type theory as well as Rocq implementation whose negative effect on performance far outweighs most potential improvements that could be achieved in the interpreter alone. The most important interactions are:

- Mtac2’s monad must be universe polymorphic but the proliferation of universe levels severely impacts overall performance of Rocq.
- Until very recently [GKY22], Rocq kernel reduction (which is also used by Mtac2’s interpreter loop) eagerly substituted universe levels in the body of cached polymorphic definitions and thereby all but negated the positive effects of the kernel’s caching mechanisms. This effectively made using universe-polymorphic definitions prohibitively expensive both in Mtac2’s interpreter as well as in general Rocq developments.⁸
- Mtac2’s interpreter cannot easily share the kernel’s cache across multiple calls to kernel reduction because the cache contains information specific to the current state of existential variables as well as the current environment. Both can change throughout the execution of Mtac2 programs, making it generally impossible to re-use the cache.

[GKY22] Gilbert et al., “Delay Universe Substitution in CClosure #15807”

⁸ At least one project relying on universe polymorphism saw an almost 50% performance increase when this issue was resolved.

1.3 STRUCTURE, PUBLICATIONS, AND COLLABORATIONS

The remainder of this thesis is split into three parts, the first two of which cover the published iGPS and Mtac2 works (Part I, and Part II, respectively). Part III goes into more detail about the challenges encountered in designing and implementing Mtac2 that could be of interest to implementers of metaprogramming and tactic languages for the Rocq proof assistant.

1.3.1 Publications

This publication contains texts from two publications:

[Kai+17] Kaiser et al., “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris”

[Kai+18] Kaiser et al., “Mtac2: typed tactics for backward reasoning in Coq”

1. “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris” [Kai+17], which is reproduced largely unchanged from the published version.
2. “Mtac2: typed tactics for backward reasoning in Coq” [Kai+18], which has been significantly improved and extended, with the largest addition being a proof of soundness.

During my PhD work I have contributed to two additional papers that are related to the projects presented in this thesis but have not been included in it. These papers are:

[Kre+18] Krebbers et al., “MoSeL: a general, extensible modal framework for interactive proofs in separation logic”

[Dan+20] Dang et al., “RustBelt meets relaxed memory”

1. “MoSeL: a general, extensible modal framework for interactive proofs in separation logic” [Kre+18], and
2. “RustBelt meets relaxed memory” [Dan+20].

1.3.2 Collaborations

The projects presented in this thesis were led by me but contain significant contributions from several co-authors. In both projects, all co-authors participated in discussions and helped with editing of the published results. Below, I list specific individual contributions.

For iGPS, I led the design of the logic, the formalization, and the writing effort. Hoang-Hai Dang contributed significantly to all parts of the projects. In terms of the formalization, he contributed the encoding of RSL rules (Section 6 and Section B) as well as the case studies (also mentioned in Section 6). In terms of writing, he contributed the introduction to Iris (Section 4), the description of additional contributions (Section 6), as well as various parts of Appendix C and Appendix B. Ori Lahav contributed the on-paper operational semantics of RA+NA (Appendix A).

For Mtac2, I led the design of typed tactics, the implementation, and the writing. Beta Ziliani contributed the encoding of dynamically-typed tactics (the `tactic` type; Section 10.1), the encoding of goals (Section 10.2), the design for tactic composition (Section 10.3), as well as an initial formulation of Mtac2’s operational semantics (Section 13, without a soundness proof). Robbert Krebbers contributed the generalized type of (dynamically-typed) tactics (the `gtactic` type; Section 10.4). Derek Dreyer spotted and helped correct multiple distinct errors in the formalization of Mtac2’s semantics.

Bibliography

- [BW05] Henk Barendregt and Freek Wiedijk. “The challenge of computer mathematics”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363.1835 (2005), pp. 2351–2375. DOI: [10.1098/rsta.2005.1650](https://doi.org/10.1098/rsta.2005.1650). eprint: <https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2005.1650>. URL: <https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2005.1650> (cit. on p. 7).
- [Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ concurrency”. In: *POPL*. ACM, 2011, pp. 55–66 (cit. on p. 5).
- [CP88] Thierry Coquand and Christine Paulin. “Inductively defined types”. In: *Conference on Computer Logic*. Vol. 417. Lecture Notes in Computer Science. Springer, 1988, pp. 50–66 (cit. on p. 6).
- [Dan+20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. “RustBelt meets relaxed memory”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 34:1–34:29 (cit. on pp. 3, 10).
- [Del00] David Delahaye. “A Tactic Language for the System Coq”. In: *LPAR*. Vol. 1955. Lecture Notes in Computer Science. Springer, 2000, pp. 85–95 (cit. on p. 7).
- [GKY22] Gaëtan Gilbert, Jan-Oliver Kaiser, and Joshua Yanovski. “Delay Universe Substitution in CClosure #15807”. 2022. URL: <https://github.com/coq/coq/pull/15807> (visited on 06/01/2024) (cit. on p. 9).
- [Kai+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris”. In: *ECOOP*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29 (cit. on p. 10).
- [Kai+18] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. “Mtac2: typed tactics for backward reasoning in Coq”. In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 78:1–78:31 (cit. on p. 10).
- [Kre+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. “MoSel: a general, extensible modal framework for interactive proofs in separation logic”. In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 77:1–77:30 (cit. on pp. 6, 10).

- [KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *POPL*. ACM, 2017, pp. 205–217 (cit. on p. 6).
- [LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. “Taming Release-acquire Consistency”. In: *POPL*. POPL 2016. ACM, 2016, pp. 649–662. ISBN: 978-1-4503-3549-2 (cit. on p. 5).
- [Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. “Repairing sequential consistency in C/C++11”. In: *PLDI*. ACM, 2017, pp. 618–632 (cit. on p. 5).
- [Lam79] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691 (cit. on p. 2).
- [LMM17] Lun Liu, Todd D. Millstein, and Madanlal Musuvathi. “A volatile-by-default JVM for server applications”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 49:1–49:25. DOI: 10.1145/3133873. URL: <https://doi.org/10.1145/3133873> (cit. on p. 5).
- [Mar+15] Daniel Marino, Todd D. Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. “The Silently Shifting Semicolon”. In: *SNAPL*. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 177–189 (cit. on p. 5).
- [Péd16] P.-M. Pédro. “Towards Ltac 2.0”. In: *DeepSpec Workshop*. 2016 (cit. on p. 7).
- [PP89] Frank Pfenning and Christine Paulin-Mohring. “Inductively Defined Types in the Calculus of Constructions”. In: *Mathematical Foundations of Programming Semantics*. Vol. 442. Lecture Notes in Computer Science. Springer, 1989, pp. 209–228 (cit. on p. 6).
- [TS18] Amin Timany and Matthieu Sozeau. “Cumulative Inductive Types In Coq”. In: *FSCD*. Vol. 108. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 29:1–29:16 (cit. on p. 6).
- [TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. “GPS: navigating weak memory with ghosts, protocols, and separation”. In: *OOPSLA*. ACM, 2014, pp. 691–707 (cit. on p. 5).
- [Wan+13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. “Towards optimization-safe systems: analyzing the impact of undefined behavior”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farmington, Pennsylvania: Association for Computing Machinery, 2013, pp. 260–275. ISBN: 9781450323888. DOI: 10.1145/2517349.2522728. URL:

<https://doi.org/10.1145/2517349.2522728> (cit. on p. 3).

- [Zil+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. “Mtac: a monad for typed tactic programming in Coq”. In: *ICFP*. ACM, 2013, pp. 87–100 (cit. on p. 8).

Part I

Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris

The field of *concurrent separation logics* (CSLs) has recently undergone two exciting developments: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Rocq, and (2) the adaptation of CSLs to account for *weak memory models*, notably C11’s release-acquire (RA) consistency. Unfortunately, these developments are seemingly incompatible, since Iris only applies to languages with an operational interleaving semantics, while C11 is defined by a declarative (axiomatic) semantics. We show that, on the contrary, it is not only feasible but useful to marry these developments together. Our first step is to provide a novel operational characterization of RA+NA, the fragment of C11 containing RA accesses and “non-atomic” (normal data) accesses. Instantiating Iris with this semantics, we then derive higher-order variants of two prominent RA+NA logics, GPS and RSL. Finally, we deploy these derived logics in order to perform the first mechanical verifications (in Rocq) of several interesting case studies of RA+NA programming. In a nutshell, we provide the first foundationally verified framework for proving programs correct under C11’s weak-memory semantics.

2

Introduction

Separation logic [Rey02] is a refinement of Hoare logic with an intrinsic notion of *ownership*: whereas an assertion in Hoare logic denotes a fact about the global machine state, an assertion in separation logic denotes ownership of (and knowledge about) a *piece* of that state, and the *separating conjunction* $P * Q$ denotes that the assertions P and Q own disjoint pieces of state. This ownership reading of assertions is useful for giving “local” (or “small-footprint”) specifications for primitive commands, which are much easier to compose soundly into specifications for larger programs. Moreover, as O’Hearn was the first to observe [OHe07], separation logic is also eminently suitable for *concurrent* programs. In particular, ownership provides a direct and convenient way of explaining how synchronization mechanisms serve to transfer control of shared state between threads. Although O’Hearn’s original concurrent version of separation logic was geared toward reasoning about coarse-grained synchronization via semaphores, the subsequent decade of research into *concurrent separation logics* (CSLs) has shown that ownership and separation are just as useful for reasoning about more fine-grained and low-level synchronization mechanisms, such as those employed in the implementations of non-blocking data structures.¹

In this chapter, we consider two boundary-pushing developments in concurrent separation logics: (1) the *Iris framework* for encoding and unifying advanced higher-order CSLs and formalizing them in Rocq,² and (2) the adaptation of CSLs to account for *weak memory models*, notably C11’s *release-acquire* (RA) consistency.³ Although these developments have thus far (for reasons explained below) appeared to be incompatible, we show that in fact they are not! Quite the contrary: we demonstrate that it is not only feasible but useful to marry them together, and in so doing, provide the first foundationally verified framework for proving programs correct under C11’s weak memory semantics.

2.1 IRIS: A UNIFYING FRAMEWORK FOR CONCURRENT SEPARATION LOGICS

After O’Hearn’s original CSL, there came a steady stream of “new and improved” CSLs appearing on at least a yearly basis. Unfortunately, as

[Rey02] Reynolds, “Separation logic: A logic for shared mutable data structures”

[OHe07] O’Hearn, “Resources, concurrency, and local reasoning”

¹ VP07; Fen09; Din+10; TDB13; SB14; Nan+14; dDG14.

² Jun+15; Jun+16; Kre+17; KTB17.

³ VN13; TVD14; DV16; LV15.

these new CSLs grew ever more expressive, they also grew increasingly complex, baking in increasingly sophisticated proof rules as primitive, with the relationships and compatibility between different proof rules (e.g., whether they could be soundly combined in one logic) remaining unclear.

The central source of complexity in most existing CSLs lies in their mechanisms for controlling *interference* between threads accessing shared state, which have evolved from Jones’s *rely-guarantee* [Jon83] to the much more sophisticated and elaborate *protocol* mechanisms appearing in logics like CaReSL [TDB13], iCAP [SB14], and TaDA [dDG14]. In an attempt to consolidate the field, Jung *et al.* developed Iris [Jun+15; Jun+16; Kre+17], a logic with the express goal of showing that even the fanciest of these interference-control mechanisms could be encoded via a combination of two orthogonal “off-the-shelf” ingredients: (1) *partial commutative monoids* (PCMs) for formalizing protocols on shared state, and (2) *invariants* for enforcing them. Invariants are an old and ubiquitous concept in program verification, and PCMs have been used in a number of prior logics to represent different kinds of *ghost* (or *auxiliary*) state *i.e.*, logical state that is manipulated as part of the proof of a program but is not manipulated directly by the program itself. Jung *et al.*’s observation was that in fact these two simple mechanisms are all you need: using just PCMs and invariants, one can *derive* a variety of powerful forms of protocol-based reasoning from prior CSLs *within* Iris, and by virtue of working in a unified framework, these derived mechanisms are automatically compatible (different mechanisms can be used soundly to verify different modules in a program). Iris also goes beyond most prior CSLs by supporting higher-order quantification and *impredicative invariants*—invariants that can talk recursively about the existence of (other) invariants—which are crucial for reasoning about languages with higher-order state (e.g., Rust).

In the past, the complexity of CSLs was further exacerbated by the fact that (before [SNB15]) they only supported manual and error-prone “pencil-and-paper” proofs. The initial version of Iris [Jun+15] was no exception: the *soundness* of the core logic was verified in Rocq, but the Rocq development provided no support for *using* the logic (either to encode other logics or to verify programs interactively). However, in 2017, Krebbers *et al.* [KTB17] have developed IPM, an interactive proof mode geared toward using Iris as a proof development environment for verifying concurrent programs within Rocq. With IPM, Iris has begun the transition to a more practically useful proof tool, and is already being deployed effectively for larger verification efforts, e.g., in the RustBelt project [Dre].

2.2 SEPARATION LOGICS FOR RELEASE-ACQUIRE CONSISTENCY

Iris is a “generic” logical framework in that it is parameterized over the programming language in question—it merely requires, like the vast ma-

[Jon83] Jones, “Tentative steps toward a development method for interfering programs”

[TDB13] Turon *et al.*, “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”

[SB14] Svendsen *et al.*, “Impredicative Concurrent Abstract Predicates”

[dDG14] da Rocha Pinto *et al.*, “TaDA: A Logic for Time and Data Abstraction”

[Jun+15] Jung *et al.*, “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”

[Jun+16] Jung *et al.*, “Higher-order ghost state”

[Kre+17] Krebbers *et al.*, “The Essence of Higher-Order Concurrent Separation Logic”

[SNB15] Sergey *et al.*, “Mechanized verification of fine-grained concurrent programs”

[KTB17] Krebbers *et al.*, “Interactive proofs in higher-order concurrent separation logic”

[Dre] Dreyer, “The RustBelt project”

jority of prior work on concurrent program verification, that the language have an *operational, interleaving semantics*, typically known as a *sequentially consistent* (SC) semantics [Lam79]. Under SC, threads take turns accessing the shared memory, and updates to memory are immediately visible to all other threads.

SC semantics has the benefit that it is easy to define and manipulate formally, but it is also woefully unrealistic: no serious language guarantees a fully SC semantics, because of the significant performance costs associated with maintaining the fiction of a single, globally consistent view of memory on modern multi-core architectures. One of the reasons for this discrepancy between the theory and the reality of concurrent programming is that, until relatively recently, formal accounts of more realistic—so-called *weak* (or *relaxed*)—memory models for concurrent programming languages were not available. However, in the past decade, great progress has been made on formalizing weak memory models, with a notable high point being the formalization of the C/C++11 memory model (hereafter, C11) [Bat+11].

In response to this development, a number of verification researchers have followed suit by building new verification tools—program logics, model checkers, testing frameworks, etc.—that account for these more realistic memory models. In particular, Vafeiadis and collaborators have thus far developed several different separation logics for C11, including RSL [VN13] and GPS [TVD14]. The main focus of these logics is on RA+NA, an important fragment of C11 consisting of *release-acquire* (RA) accesses and *non-atomic* (NA) accesses. RA accesses are useful because they support a common idiom of message-passing synchronization at low cost compared to SC. NA accesses are intended for “normal” data accesses and are even more efficiently implementable than RA accesses, with the proviso that they are not permitted to race (*i.e.*, races on non-atomics cause the entire program to have undefined behavior).

A major challenge that Vafeiadis *et al.* had to overcome was the fact that C11 is defined using a radically different semantics than SC. Specifically, it is defined by a *declarative* (or *axiomatic*) semantics, in which the allowed behaviors of a program are defined by enumerating candidate executions (represented as “event graphs”) and then restricting attention to the executions that obey various coherence axioms. In building separation logics for C11, Vafeiadis *et al.* were thus not able to use the standard model of Hoare-style program specifications from prior separation logics because notions like “the machine states before and after executing a command C ” do not have a clear meaning in C11’s declarative semantics.

To account for this radically different type of semantics, they were instead forced to essentially throw away the “separation-logic textbook” and come up with an entirely new, non-standard model of separation logic in terms of predicates on event graphs. While groundbreaking, this approach has had several downsides. Firstly, certain essential mechanisms of SC-based separation logic (such as ghost state), which are easy to justify in standard models, became very difficult to justify in the new event-graph-based models of RA+NA logics. Secondly, the complexity of these new models has made them challenging to adapt and extend,

[Lam79] Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”

[Bat+11] Batty et al., “Mathematizing C++ concurrency”

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A program logic for C11 concurrency”

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”

and their non-standard nature has posed a major accessibility hurdle for researchers accustomed to traditional models of separation logic. Last but not least, although the soundness of these logics has been verified formally in Rocq, there has thus far been no tool support for *using* the logics to prove programs correct under RA+NA semantics.

2.3 OUR CONTRIBUTIONS

Given our above description, it may seem that the Iris framework’s reliance on interleaving semantics renders it fundamentally inapplicable to reasoning about C11’s weak-memory semantics. We show that this is not the case at all—not only is it possible to derive RA+NA logics like GPS and RSL within Iris, but there are several tangible benefits to doing so. Deriving such logics within Iris:

- Lets us take advantage of the rich features of the Iris host logic (*e.g.*, separation, invariants) when proving soundness of the derived logics, thereby significantly lifting the abstraction level at which those soundness proofs are carried out (compared to prior work).
- Allows us to support some very useful features in our derived logics by directly importing them from Iris. Such features include PCM-based ghost state, higher-order impredicative quantification, and Iris’s interactive proof mode in Rocq. By virtue of being encoded in Iris, our derived logics inherit these features for free.
- Makes it easy to experiment with the derived logics and quickly develop new and useful extensions (*e.g.*, single-writer protocols, see below).
- Makes it possible to soundly compose proofs from different derived logics, since they are all carried out in the uniform framework of Iris.

Our first step (§3) is to avoid the essential complicating factor—C11’s declarative account of RA+NA—and instead work with an operational account. Building closely on Lahav *et al.*’s proposed “strong release-acquire” (SRA) semantics [LGV16], we define a novel, operational, interleaving semantics for RA+NA. Our operational account of the RA fragment of the language is very similar to Lahav *et al.*’s operational account of SRA in that it models writing and reading of memory via the sending and receiving of timestamped messages; the main difference is that the RA rule for assigning timestamps is slightly more liberal. Our account of NA is new, though; it uses timestamps to model races on non-atomics as stuck (unsafe) machine states. We have proven that, under the reasonable restriction that programs do not mix RMWs (atomic updates) and non-atomic reads at the same location, our semantics is equivalent to the standard declarative semantics of the RA+NA fragment of C11.

Next, since our new semantics for RA+NA is an interleaving semantics, we can instantiate Iris with it. In §4, we review the basic reasoning mechanisms of Iris, and show how to use them to derive small-footprint proof rules for reasoning about RA+NA programs. We apply these rules to

[LGV16] Lahav et al., “Taming Release-acquire Consistency”

verify a simple message-passing example of RA+NA programming in Iris. However, as will become clear, reasoning directly with the Iris primitive mechanisms is rather too low-level and a more abstract logic is needed.

In §5, we present iGPS, a higher-order variant of Turon *et al.*'s GPS logic [TVD14], which supports much higher-level reasoning about RA+NA programs. Unlike the original GPS, iGPS is derived within Iris on top of the small-footprint proof rules from §4. It also extends GPS with *single-writer protocols*, an extremely useful feature that simplifies proofs of RA+NA programs in the common case where there are no write-write races on atomic accesses.

In §6, we briefly describe some other contributions, including iRSL, a higher-order variant of RSL [VN13] derived within Iris, and several case studies that we have verified using iGPS and iRSL in Rocq. These examples showcase one of the major advantages of working in the Iris framework: our ability to verify weak-memory programs, foundationally and mechanically, with the same degree of ease that was previously only possible for SC programs.

Finally, in §7, we conclude with related work.

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A program logic for C11 concurrency”

3

Release-Acquire and Non-Atomics

In this section, we introduce our operational semantics for RA+NA, which we then use as the machine for our working language λ_{RN} . Subsequent sections will show how to build a logic for λ_{RN} using Iris.

C11 provides several memory access modes, each ensuring a different degree of consistency. In this thesis, we focus on RA+NA, the fragment of C11 consisting only of release-acquire (RA) and non-atomic (NA) accesses. Non-atomic accesses (which we denote with “[na]”) are the default type of memory accesses, intended to be used for normal data rather than for synchronization. Thus, C11 forbids any data races on non-atomic accesses, and programs that may have such races are considered buggy (they have undefined semantics). In contrast, RA accesses (which we denote with “[at]” for *atomic*) are permitted to race, but provide just enough consistency guarantees to enable the well-known *message passing* (MP) idiom:

$$\begin{array}{l} x_{[\text{na}]} := 0; y_{[\text{na}]} := 0; \\ x_{[\text{na}]} := 37; \quad \left\| \begin{array}{l} \textbf{repeat } y_{[\text{at}]}; \\ y_{[\text{at}]} := 1 \end{array} \right. \quad \left\| \quad x_{[\text{na}]} \end{array}$$

Initially, both variables x and y are set to 0. The first thread will initialize x to 37 (non-atomically) and then set the variable y to 1 (via a release write) as a way of sending a message to the second thread that x has been properly initialized and is ready for consumption. The second thread will repeatedly read y (via an acquire read) until it observes $y \neq 0$, at which point—thanks to release-acquire semantics—it will know that it can safely access x . Summing up, the use of RA here ensures that the non-atomic write to x in the first thread “happens before” the non-atomic read of x in the second thread—*i.e.*, that they do not race—and furthermore that the read of x will return 37.

The formal semantics of RA+NA is “declarative”, formulated as a set of constraints on execution graphs. We will instead now present an alternative *operational* semantics of RA+NA. Our operational semantics is not completely coherent with C11’s for programs that mix atomic and non-atomic accesses to the same location (although the semantics of such programs is already known to be problematic [Bat+15]—see §7 for further discussion of this point). However, for the large class of programs

[Bat+15] Batty et al., “The Problem of Programming Language Concurrency Semantics”

that do not mix atomic updates (like CAS) and non-atomic reads at the same location, our semantics is provably equivalent to C11’s declarative semantics. This class of programs includes all C11 programs considered (and verified) in this thesis. (For formal details of the correspondence between our semantics and C11’s, see [Appendix A](#).) We will first start with the pure RA fragment, and then add a “race detector” for non-atomic accesses.

3.1 RELEASE-ACQUIRE

Our operational semantics for RA starts from the observation that in RA—in contrast to a standard heap language—different threads have a different view of what the state is. Accordingly, we need to keep track of past write events as they might still be relevant for some subset of threads. Moreover, we need to keep writes to the same location in a total order enforced in C11 under the name *modification order* (*mo* for short). Finally, we also need to keep track of each thread’s “progress” in terms of which writes are visible to it, as this determines what a thread may read and where its writes may end up.

For the *mo* order, the RA machine manages for each location a totally ordered set of timestamps $t \in \text{Time} \triangleq \mathbb{N}$. Each write of some value v to a location ℓ gets assigned a timestamp (that is unique for ℓ), resulting in a write event $\omega \in \text{Event} \triangleq \text{Loc} \times \text{Val} \times \text{Time}$, where $v \in \text{Val} \triangleq \mathbb{Z}$.¹ Using timestamps, the thread’s “progress” is represented by a *view*, $V \in \text{View} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Time}$, which records the timestamp of the most recent write event observed by the thread for every location. To enable communication between threads, every write event is augmented with the writing thread’s view, yielding a message $m \in \text{Msg} \triangleq \text{Event} \times \text{View}$. The machine state σ comprises a message pool (called *memory*) and a view for every thread.

¹ The full semantics also supports allocation, which induces an allocation value A . We do not mention it here for the sake of simplicity.

Definition 3.1.1 (Simplified Physical State). *Let*

$$\sigma \in \Sigma \triangleq \left(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View}) \right) \uplus \{ \perp_{\text{uninit}} \}$$

represent physical machine states, where $\perp_{\text{uninit}}$ represents an error state. We write M and T to denote the two components of a non-error state.

The λ_{RN} language’s reductions are factored into *expression reductions*, concerned with the evaluation of the language’s expressions, and *machine reductions*, concerned with how the execution of an expression affects the machine state. We will define the expression reductions later when we formally define λ_{RN} . Here we focus on the machine reductions.

We define event labels $\varepsilon \in \mathcal{E} \triangleq \{ \langle \text{Read}, \ell, v \rangle, \langle \text{Write}, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle, \langle \text{Fork}, \rho \rangle \}$, representing reads, writes, atomic updates (RMW’s), and forks (with ρ being the newly created thread id), respectively. The reductions are defined by a set of local, per-thread reductions $\xrightarrow{\varepsilon}^\pi \subseteq \Sigma \times \Sigma$ given in [Figure 3.1](#), where π represents the current thread’s id.

A write (**THREAD-WRITE**) picks an *unused* timestamp t for location ℓ that is greater than the thread’s view of ℓ , updates the thread’s view to

$$\begin{array}{c}
\text{Thread-Read} \\
\frac{(\ell, v, t, V) \in M \quad T(\pi)(\ell) \leq t}{(M, T) \xrightarrow{\langle \text{Read}, \ell, v \rangle}^\pi (M, T[\pi \mapsto T(\pi) \sqcup V])} \\
\\
\text{Thread-Write} \\
\frac{\neg \exists v', V. (\ell, v', t, V) \in M \quad T(\pi)(\ell) < t \quad V' = T(\pi)[\ell \mapsto t]}{(M, T) \xrightarrow{\langle \text{Write}, \ell, v \rangle}^\pi (M \cup \{(\ell, v, t, V')\}, T[\pi \mapsto V'])} \\
\\
\text{Thread-Update} \\
\frac{(\ell, v_o, t, V) \in M \quad T(\pi)(\ell) \leq t \quad \neg \exists v, V. (\ell, v, t+1, V) \in M \quad V' = T(\pi)[\ell \mapsto t+1] \sqcup V}{(M, T) \xrightarrow{\langle \text{Update}, \ell, v_o, v_n \rangle}^\pi (M \cup \{(\ell, v_n, t+1, V')\}, T[\pi \mapsto V'])} \\
\\
\text{Thread-Fork} \quad \text{Thread-Uninitialized} \\
\frac{\rho \notin \text{dom}(T)}{(M, T) \xrightarrow{\langle \text{Fork}, \rho \rangle}^\pi (M, T[\rho \mapsto T(\pi)])} \quad \frac{\begin{array}{c} \varepsilon \in \{ \langle \text{Read}, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \\ T(\pi)(\ell) = \perp \end{array}}{(M, T) \xrightarrow{\varepsilon}^\pi \perp_{\text{uninit}}}
\end{array}$$

Figure 3.1: Per-thread reductions for RA without NA.

the new view V' that includes t , and adds a corresponding message to the memory. A read (**THREAD-READ**) incorporates the view V of the message that it reads into the thread's own view.² Note that the message being read is required to have a timestamp that is not smaller than the thread's view of the relevant location. Updates (**THREAD-UPDATE**) combine reading and writing in one step. In addition, updates must “pick” $t+1$ as a timestamp for the new message, where t is the timestamp of the read message. This implies, in particular, that two different updates cannot read the same message, and corresponds to C11's atomicity condition, which requires every update to read from its *mo*-immediate predecessor. **THREAD-FORK** adds a new thread whose view is copied from its parent. Finally, **THREAD-UNINITIALIZED** detects reads from uninitialized locations, and moves to the error state $\perp_{\text{uninit}}$.

² Here, we use the join operator $\sqcup$ on views: $(V_1 \sqcup V_2)(\ell) = \max\{V_1(\ell), V_2(\ell)\}$ if $\ell \in \text{dom}(V_1) \cap \text{dom}(V_2)$; $(V_1 \sqcup V_2)(\ell) = V_i(\ell)$ if $\ell \in \text{dom}(V_i) \setminus \text{dom}(V_j)$; and $(V_1 \sqcup V_2)(\ell)$ is undefined if $\ell \notin \text{dom}(V_1) \cup \text{dom}(V_2)$.

Functional correctness of MP

With the operational semantics of RA, we can now sketch why MP (assuming for now that all its accesses are RA) is functionally correct, *i.e.*, why the read of x by the second thread will return 37 when the program terminates. The write of 37 to x is recorded at a view V_{37} , which is then included in the view V_1 of the write of 1 to y by the first thread. When the second thread reads 1 from y , its local view is updated to incorporate V_1 (and thus also V_{37}). A read from x is now guaranteed to read from the message setting x to 37 or from a more recent one, but no more recent one exists. Consequently, the return value will be 37.

3.2 NON-ATOMICS

Formally, C11 defines a data race as two memory accesses to the same location—of which at least one is a write and at least one is non-atomic—

Figure 3.2: Per-thread reductions for the RA+NA machine.

$$\begin{array}{c}
\text{Read} \\
\frac{(\ell, v, t, V) \in M \quad T(\pi)(\ell) \leq t \quad \alpha = \text{na} \Rightarrow \forall v', t', V'. (\ell, v', t', V') \in M \Rightarrow t' \leq T(\pi)(\ell)}{(M, T, N) \xrightarrow{\langle \text{Read}_\alpha, \ell, v \rangle}^\pi (M, T[\pi \mapsto T(\pi) \sqcup V], N)} \\
\\
\text{Write-at} \\
\frac{\neg \exists v', V. (\ell, v', t, V) \in M \quad N(\ell) \leq T(\pi)(\ell) < t \quad V' = T(\pi)[\ell \mapsto t]}{(M, T, N) \xrightarrow{\langle \text{Write}_{\text{at}}, \ell, v \rangle}^\pi (M \cup \{(\ell, v, t, V')\}, T[\pi \mapsto V'], N)} \\
\\
\text{Write-na} \\
\frac{\neg \exists v', V. (\ell, v', t, V) \in M \quad N(\ell) \leq T(\pi)(\ell) \quad V' = T(\pi)[\ell \mapsto t] \quad \forall v', t', V. (\ell, v', t', V) \in M \Rightarrow t' < t}{(M, T, N) \xrightarrow{\langle \text{Write}_{\text{na}}, \ell, v \rangle}^\pi (M \cup \{(\ell, v, t, V')\}, T[\pi \mapsto V'], N[\ell \mapsto t])} \\
\\
\text{Update} \\
\frac{(\ell, v_o, t, V) \in M \quad N(\ell) \leq T(\pi)(\ell) \leq t \quad \neg \exists v, V. (\ell, v, t+1, V) \in M \quad V' = T(\pi)[\ell \mapsto t+1] \sqcup V}{(M, T, N) \xrightarrow{\langle \text{Update}, \ell, v_o, v_n \rangle}^\pi (M \cup \{(\ell, v_n, t+1, V')\}, T[\pi \mapsto V'], N)} \\
\\
\text{Fork} \\
\frac{\rho \notin \text{dom}(T)}{(M, T, N) \xrightarrow{\langle \text{Fork}, \rho \rangle}^\pi (M, T[\rho \mapsto T(\pi)], N)} \\
\\
\text{Race-I} \\
\frac{\varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Write}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \quad T(\pi)(\ell) < N(\ell)}{(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}} \\
\\
\text{Race-II} \\
\frac{\exists v', t', V'. (\ell, v', t', V') \in M \wedge T(\pi)(\ell) < t'}{(M, T, N) \xrightarrow{\langle \text{Read}_{\text{na}}, \ell, v \rangle}^\pi \perp_{\text{race}}} \\
\\
\text{Uninitialized} \\
\frac{\varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \quad T(\pi)(\ell) = \perp}{(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{uninit}}}
\end{array}$$

that are not ordered by “happens-before.” A program that exhibits data races in *some* of its execution graphs is called *racy*, and its behavior is considered undefined. We now show how to account for non-atomics and data races in the context of our operational semantics.

Let us first extend the set of physical states by another error state $\perp_{\text{race}}$, whose intent is captured by the following correspondence: a program is *racy* if and only if at least one of its machine executions can reach $\perp_{\text{race}}$ (stated and proved formally in [Appendix A](#)).

To detect data races during the execution of a program, we add an additional component to the physical state: the *non-atomic view* N , which tracks the timestamp of the most recent non-atomic write to every location. Then, we place the following restrictions on all atomic and non-atomic operations (if violated, the program will enter the $\perp_{\text{race}}$ state):

- To perform any access (atomic or non-atomic) to a location ℓ , a thread

π must have observed the most recent non-atomic write to ℓ , i.e., $N(\ell) \leq T(\pi)(\ell)$.

- A thread π can only perform a non-atomic read from a location ℓ if it has observed the most recent (atomic or non-atomic) write to ℓ , i.e., $\nexists t, (\ell, _, t, _) \in M. T(\pi)(\ell) < t$.

In addition to these restrictions, we require non-atomic writes to pick timestamps greater than all existing timestamps of messages of the same location. Intuitively, these restrictions enforce that each non-atomic write to ℓ starts a new “era” in ℓ ’s timestamps, after which any attempt to access writes from a previous era (or to write with a timestamp from a previous era) constitutes a race. Note that there is an asymmetry between non-atomic reads and writes: non-atomic writes to ℓ are allowed even when the thread has not observed the most recent write to ℓ , as it is only required to observe the most recent *non-atomic* write to ℓ . One might fear that this fails to detect the case when a non-atomic write is racing with a concurrent atomic write (and the atomic write happens first); but in this case the race will be detected in a different execution where the non-atomic write happens first (and the atomic write enters the $\perp_{\text{race}}$ state), so the program will nevertheless be declared racy.

Revisiting MP, we note that it is safe to have non-atomic accesses to x : the write is performed while the left thread is necessarily aware of the most recent non-atomic write to x (the initialization); and the read is performed while the right thread is necessarily aware of the most recent write to x , whose timestamp was incorporated into the right thread’s view when it read $y = 1$.

Figure 3.2 presents the full operational semantics. It is based on the following definition of a physical state.

Definition 3.2.1 (Physical State).

Let $\sigma \in \Sigma \triangleq \left(\mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View}) \times \text{View} \right) \uplus \{ \perp_{\text{race}}, \perp_{\text{unit}} \}$ represent physical machine states. We write M , T , and N to denote the components of a non-error state. The initial physical state, denoted σ_{init} , is given by $(\emptyset, [0 \mapsto \emptyset], \emptyset)$.

3.3 THE λ_{RN} LANGUAGE

λ_{RN} is a standard lambda calculus with recursive functions, forks, and references with atomic and non-atomic accesses. The **repeat** construct that we have used in MP can be defined in terms of recursive functions. The interesting part of the language and its expression reductions is given in Figure 3.3. The expression reduction relation $(e \xrightarrow{\varepsilon} e', \bar{e}_f)$ has four components: the original expression e , an (optional) machine memory event ε , the resulting expression e' , and a list of newly created threads $\bar{e}_f$. Only the rule for **fork** e creates a new thread (i.e., a singleton list $[e]$), while all other reductions produce an empty list (i.e., nil).

The *per-thread language reductions* $(\sigma; e \xrightarrow{\varepsilon} \sigma'; e', \bar{e}_f)$ are then the combination of the expression reductions and the machine reductions,

Figure 3.3: Main λ_{RN} expressions and expression reductions.

$$\begin{aligned}
v \in \text{Val} &::= () \mid z \in \mathbb{Z} \mid \ell \in \text{Loc} \mid \mathbf{fix} (f, x). e \\
\alpha \in \text{Access} &::= \text{at} \mid \text{na} \\
e \in \text{Expr} &::= v \mid e_1 e_2 \mid \mathbf{if} e \mathbf{then} e_1 \mathbf{else} e_2 \mid \\
&\quad \mid \mathbf{fork} e \mid e_{[\alpha]} \mid e_{1[\alpha]} := e_2 \mid \mathbf{cas}(\ell, e_1, e_2) \mid \dots
\end{aligned}$$

$$\begin{array}{lll}
\ell_{[\alpha]} & \xrightarrow{\langle \text{Read}_{\alpha}, \ell, v \rangle} & v, \text{nil} \\
\ell_{[\alpha]} := v & \xrightarrow{\langle \text{Write}_{\alpha}, \ell, v \rangle} & (), \text{nil} \\
\mathbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Update}, \ell, v_o, v_n \rangle} & 1, \text{nil} \\
\mathbf{cas}(\ell, v_o, v_n) & \xrightarrow{\langle \text{Read}_{\text{at}}, \ell, v \rangle} & 0, \text{nil} \quad \text{if } v \neq v_o \\
\mathbf{fork} e & \xrightarrow{\langle \text{Fork}, \rho \rangle} & (), [e] \\
(\mathbf{fix} (f, x). e) v & \rightarrow & e[(\mathbf{fix} (f, x). e)/f][v/x], \text{nil} \\
\mathbf{if} z \mathbf{then} e_1 \mathbf{else} e_2 & \rightarrow & e_1, \text{nil} \quad \text{if } z \neq 0 \\
\mathbf{if} z \mathbf{then} e_1 \mathbf{else} e_2 & \rightarrow & e_2, \text{nil} \quad \text{if } z = 0 \\
& \dots &
\end{array}$$

given by the COMBINED-* rules in Figure 3.4. Non-stateful reductions (COMBINED-PURE) simply defer to the expression reductions, while stateful reductions (COMBINED-MEM and COMBINED-FORK) use the event label ε and the thread id π to tie the expression and machine reductions together correctly. These per-thread reductions then are lifted in a straightforward manner to the full (threadpool) reductions.

Figure 3.4: Threadpool reductions.

$$\begin{array}{ll}
\text{Combined-Pure} & \text{Combined-Mem} \\
\frac{e \rightarrow e', \text{nil}}{\sigma; e \rightarrow^{\pi} \sigma; e', \text{nil}} & \frac{\begin{array}{l} \varepsilon \neq \langle \text{Fork}, _ \rangle \\ e \xrightarrow{\varepsilon} e', \text{nil} \quad \sigma \xrightarrow{\varepsilon}^{\pi} \sigma' \end{array}}{\sigma; e \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \text{nil}} \\
\\
\text{Combined-Fork} & \\
\frac{e \xrightarrow{\langle \text{Fork}, \rho \rangle} e', [e_f] \quad \sigma \xrightarrow{\langle \text{Fork}, \rho \rangle}^{\pi} \sigma'}{\sigma; e \xrightarrow{\langle \text{Fork}, \rho \rangle}^{\pi} \sigma'; e', [e_f]} & \\
\\
\text{Threadpool-Red-Pure} & \text{Threadpool-Red-Mem} \\
\frac{\sigma; \mathcal{TS}(\pi) \rightarrow^{\pi} \sigma'; e', \text{nil}}{\sigma; \mathcal{TS} \rightarrow^{\pi} \sigma'; \mathcal{TS}[\pi \mapsto e']} & \frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\varepsilon}^{\pi} \sigma'; e', \text{nil}}{\sigma; \mathcal{TS} \xrightarrow{\varepsilon}^{\pi} \sigma'; \mathcal{TS}[\pi \mapsto e']} \\
\\
\text{Threadpool-Red-Fork} & \\
\frac{\sigma; \mathcal{TS}(\pi) \xrightarrow{\langle \text{Fork}, \rho \rangle}^{\pi} \sigma'; e', [e_f]}{\sigma; \mathcal{TS} \xrightarrow{\langle \text{Fork}, \rho \rangle}^{\pi} \sigma'; \mathcal{TS}[\pi \mapsto e'] \uplus [\rho \mapsto e_f]} &
\end{array}$$

4

Iris

Iris is a generic framework for constructing concurrent separation logics. One can instantiate the framework with any language that has an operational interleaving semantics, and then easily derive time-tested reasoning principles for one’s target logic, including various “protocol” mechanisms for controlling interference. [Figure 4.1](#) provides an excerpt of Iris syntax.

$$P ::= \top \mid \perp \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \multimap Q \mid \exists x. P \mid \forall x. P \mid \mu x. P \\ \mid \text{Phys}(\sigma) \mid [\bar{a}]^\gamma \mid \triangleright P \mid \Box P \mid \boxed{P}^\mathcal{N} \mid P \stackrel{\mathcal{N}_1}{\Rightarrow} \mathcal{N}_2 Q \mid \{P\} e \{x. Q\}_\mathcal{N} \mid \dots$$

Figure 4.1: An excerpt of Iris syntax.

Iris supports the common connectives ($\top, \perp, \Rightarrow, \wedge, \vee, *, \multimap, \exists, \forall, \mu$) and proof rules standard in higher-order separation logics. Iris’s extended set of constructs includes physical state ownership $\text{Phys}(\sigma)$, ghost state ownership $[\bar{a}]^\gamma$, the later $\triangleright$ and always $\Box$ modalities, invariants $\boxed{P}^\mathcal{N}$, view shifts $P \stackrel{\mathcal{N}_1}{\Rightarrow} \mathcal{N}_2 Q$, and Hoare triples $\{P\} e \{x. Q\}_\mathcal{N}$. We will first explain these constructs via a running example, in which we use Iris to verify the MP example in a simple, sequentially consistent language called λ_{SC} . This will not only illustrate how one can derive within Iris a target logic for a language defined by an operational semantics, but will also serve as a warm-up for our subsequent explanation of how we can instantiate Iris to reason about weak memory.

Road map

The process of instantiating Iris to derive new logics follows a simple pattern, which is worth articulating up front:

1. When we first instantiate Iris, the only primitive assertion we get about the state of the program is the *physical state ownership* assertion $\text{Phys}(\sigma)$, which asserts that σ is the current global state of the machine. Together with this assertion we also get for free a bunch of *large-footprint* specifications for the primitive commands of the language, based directly on their operational semantics. For example, the primitive specification we get for updating a location in λ_{SC} will be $\{\text{Phys}(\sigma)\} \ell := v \{\text{Phys}(\sigma[\ell \mapsto v])\}$.
2. Of course, one of the main points of separation logic is to be able to reason modularly using *local* assertions about the machine state,

such as the *points-to* assertion, $\ell \hookrightarrow v$, and correspondingly give *small-footprint* specifications of the primitive commands, such as $\{\ell \hookrightarrow w\} \ell := v \{\ell \hookrightarrow v\}$. In Iris, such local assertions are not baked into the logic, but rather are encodable using *ghost state ownership* assertions, and the user of the logic has a great deal of flexibility concerning how these assertions are defined. In the case of the *points-to* assertion, $\ell \hookrightarrow v$, we will define this assertion so as to represent the *knowledge* that ℓ currently points to v and the *rights* to read and write ℓ .

3. On its own, a local, user-defined ghost state assertion like the *points-to* assertion is merely a *representation* of knowledge and rights. In order to give meaning to such a ghost state assertion—*i.e.*, to make sure it is in sync with the primitive physical state assertion—we establish an *invariant* tying the assertions together. In the case of the *points-to* assertion, this invariant will enforce that when a thread owns the ghost state assertion $\ell \hookrightarrow v$, its “knowledge” that ℓ currently points to v in the physical machine state is actually correct.

In short, *ghost state assertions* represent local knowledge and rights concerning the machine state, and *invariants* enforce that ghost state assertions mean what they say they mean.

4.1 IRIS BY EXAMPLE

Figure 4.2: Main heap-related reductions of the λ_{SC} language.

$$\begin{array}{lll} (!\ell, \sigma) & \rightarrow & (v, \sigma) \quad \text{if } \sigma(\ell) = v \\ (\ell := v, \sigma) & \rightarrow & ((), \sigma[\ell \mapsto v]) \quad \text{if } \ell \in \text{dom}(\sigma) \\ & \dots & \end{array}$$

Our example programming language λ_{SC} is a standard lambda calculus with references. It is basically the same as λ_{RN} , except that all accesses are sequentially consistent, and races are permitted (they do not induce stuckness). The language’s physical state is a heap σ , which is a finite map from allocated locations to values. The main heap-related reductions of λ_{SC} are given in [Figure 4.2](#). When we instantiate Iris with λ_{SC} ’s operational semantics, (as explained in the above road map) what we get automatically from Iris are the following *large-footprint* Hoare triples concerning the physical state ownership assertion $\text{Phys}(\sigma)$:

$$\begin{array}{l} \text{Phys-Heap-Read} \\ \{\text{Phys}(\sigma) * \sigma(\ell) = v\} !\ell \{z. z = v * \text{Phys}(\sigma)\} \\ \\ \text{Phys-Heap-Write} \\ \{\text{Phys}(\sigma)\} \ell := v \{\text{Phys}(\sigma[\ell \mapsto v])\} \end{array}$$

Note that z in the first triple binds the return value of the expression $!\ell$. In the second triple, the expression returns the unit value, so we elide the binder.

4.1.1 Encoding Separation Logic for λ_{SC}

We would now like to encode these *small-footprint* Hoare triples for λ_{SC} :

$$\begin{array}{ll} \text{Heap-Read} & \text{Heap-Write} \\ \{\ell \hookrightarrow v\} !\ell \{z. z = v * \ell \hookrightarrow v\} & \{\ell \hookrightarrow w\} \ell := v \{\ell \hookrightarrow v\} \end{array}$$

The first step is to define the points-to assertion, $\ell \hookrightarrow v$, using Iris's *ghost state*.

Ghost state and partial commutative monoids

Ghost state is non-physical state that is only used as part of a program verification but is not itself part of the machine state. In Iris, ghost state is formalized using partial commutative monoids (PCMs).¹ The assertion $\llbracket a : M \rrbracket^\gamma$ asserts the ownership of the ghost resource a for an instance γ of the PCM M . Separating conjunction for ghost state assertions simply lifts the PCM composition operation to the assertion level: $\llbracket a : M \rrbracket^\gamma * \llbracket b : M \rrbracket^\gamma \iff \llbracket a \cdot_M b : M \rrbracket^\gamma$. If two PCM fragments are not compatible (i.e. their composition is not defined), then it is not possible to own both of them at the same time, i.e., if $a \cdot b = \perp$ then $\llbracket a \rrbracket^\gamma * \llbracket b \rrbracket^\gamma \Rightarrow \top$.² In order to maintain consistency of the logic, therefore, changes to ghost state are restricted to *frame-preserving* updates, in which a PCM fragment a can only be updated to b if b preserves compatibility with any other fragments in the environment (the *frame*):

$$\frac{\text{Ghost-Update} \quad \forall a_f. a \cdot a_f \neq \perp \Rightarrow b \cdot a_f \neq \perp}{\llbracket a \rrbracket^\gamma \Rightarrow \llbracket b \rrbracket^\gamma}$$

Ghost updates belong to the set of *logical computations*, or in Iris terminology, *view shifts*. A view shift $P \Rightarrow Q$ represents the capability of transforming a resource satisfying P into a resource satisfying Q without changing the physical state.

A PCM for heaps

As a step towards defining $\ell \hookrightarrow v$, let us now construct a PCM called **HEAP** that has the same basic structure as the physical heap, but allows splitting and recomposition. (We will ultimately need a slightly more sophisticated PCM to define $\ell \hookrightarrow v$, but **HEAP** is an important part of the construction.) **HEAP** is a finite partial map from locations to values, with the empty heap as its unit element, and the composition on heaps is defined as disjoint union (i.e., union if the heaps have disjoint domain, and undefined otherwise). The composition implies that the singleton heap $[\ell := v]$ does not combine with itself, so it can only be uniquely owned, and it also represents the permission required to update ℓ :

$$\begin{array}{ll} \text{Ghost-Heap-Exclusive} & \text{Ghost-Heap-Update} \\ \llbracket [\ell := v] \rrbracket^\gamma * \llbracket [\ell := w] \rrbracket^\gamma \vdash \top & \llbracket [\ell := w] \rrbracket^\gamma \Rightarrow \llbracket [\ell := v] \rrbracket^\gamma \end{array}$$

¹ Actually, ghost state in Iris is based on the more general mechanism of “cameras” (aka step-indexed resource algebras), which can support a more general form of higher-order ghost state [Jun+16].

² From here on we also suppress the PCM M in $\llbracket a : M \rrbracket^\gamma$ when it can be inferred in context.

The singleton heap $[\ell := v]$ therefore has the desired properties for defining the local assertion $\ell \hookrightarrow v$, but unfortunately it is still not quite enough: we also need some way to tie this ghost state assertion to the underlying physical state of the program. Toward this end, we employ Iris’s *invariants*.

Invariants

Invariants in Iris can be thought of as assertions that hold of *some* shared resource at all times, although the choice of which shared resource satisfies them is allowed to vary over time. The Iris invariant assertion $\boxed{P}^{\mathcal{N}}$ stipulates that P is an invariant. The resource that satisfies it is shared with all threads, and thus any thread can access it freely in a single physical step³: it can *open* the invariant and gain local ownership of the resource for the duration of the operation, so long as it can *close* the invariant by relinquishing ownership of some (potentially different) resource satisfying P at the end of the operation. For bookkeeping purposes—specifically, to ensure that we do not unsoundly open the same invariant more than once in a nested fashion—invariants in Iris are named, and the $\mathcal{N}$ in the above invariant assertion is a *namespace* (set of names) from which the name of the invariant must come.

³In Iris terminology, a resource in an invariant can be accessed within an *atomic* operation, which is an operation that takes only a single physical step of execution. We do not use the term here to avoid confusion with C11 atomics.

Invariants belong to the set of *persistent* assertions, denoted with the always modality $\Box$. The assertion $\Box P$ establishes the *knowledge* that P holds without any ownership, and therefore holds forever after. Putting resources into an invariant is thus a common way to share or transfer ownership through the use of freely distributable knowledge.

Meanwhile, the actions of opening and closing invariants belong to the set of logical computations, or view shifts. To account for invariants, view shifts are extended with namespaces as well: $P \stackrel{\mathcal{N}_1}{\Rightarrow} \mathcal{N}_2 Q$ asserts that, assuming the invariants named in $\mathcal{N}_1$ hold before the view shift, then the invariants named in $\mathcal{N}_2$ hold after the view shift. Opening and closing of invariants are then formalized as follows:

$$\begin{array}{ll} \text{Inv-Open} & \text{Inv-Close} \\ \boxed{P}^{\mathcal{N}} \vdash \perp \stackrel{\mathcal{N} \Rightarrow \emptyset}{\Rightarrow} P & \boxed{P}^{\mathcal{N}} \vdash P \stackrel{\emptyset \Rightarrow \mathcal{N}}{\Rightarrow} \perp \end{array}$$

INV-OPEN allows a thread to open the invariant $\boxed{P}^{\mathcal{N}}$ and gain ownership of P but prevents it from doing so more than once. Only after applying **INV-CLOSE** and re-establishing the invariant will the thread be able to open it again.

Note: The **INV-OPEN** rule as stated here is only sound if P talks about ownership (of physical or ghost state) and not about the existence of other invariants. In general, however, Iris makes no such restriction; rather, it supports *impredicative invariants*, meaning that P can be an arbitrary assertion. In order to avoid paradoxes caused by impredicative circularities (like the one described in [Kre+17]), the fully general version of this rule in Iris requires that P be guarded by the step-indexed *later* modality ($\triangleright$). Fortunately, in most cases the $\triangleright$ ’s can be stripped away automatically (the Iris proof mode in Rocq provides support for doing this)

[Kre+17] Krebbers et al., “The Essence of Higher-Order Concurrent Separation Logic”

and do not play an interesting role in proofs. To focus the presentation, we will therefore suppress further discussion of the $\triangleright$ modality.

Hoare triples in Iris are also annotated with invariant namespaces, since Hoare triples combine both physical and logical computations. A Hoare triple $\{P\} e \{x. Q\}_{\mathcal{N}}$ with a namespace $\mathcal{N}$ implies that if the invariants in $\mathcal{N}$ hold before the expression's execution, then they will be preserved between every step and also after its execution. Consequently, when verifying any single physical step of computation, we are free to open the invariants in $\mathcal{N}$ so long as we immediately close them. This reasoning is encapsulated in the following “atomic rule of consequence”:

$$\text{AConsq} \quad \frac{\{P'\} e \{v. Q'\}_{\mathcal{N}} \quad \forall v. Q' \stackrel{\mathcal{N}}{\Rightarrow} \stackrel{\mathcal{N} \uplus \mathcal{N}'}{\Rightarrow} Q \quad e \text{ takes 1 physical step}}{\{P\} e \{v. Q\}_{\mathcal{N} \uplus \mathcal{N}'}}$$

Since bookkeeping of namespaces is largely a tedious detail (and one which Rocq will force us to get right), we will for the remainder of this chapter suppress namespaces from definitions and proofs. We will always use disjoint namespaces to ensure correctness in opening invariants.

Linking physical and ghost state using invariants and the “authoritative” PCM

Now, returning to our example, the key idea is to use an invariant to tie the physical state assertion together with local ghost state assertions, thereby giving them meaning. To achieve this, we will employ an extremely useful construction called the *authoritative* PCM [Jun+15].

Given a base PCM $\mathcal{M}$, the authoritative PCM $\text{AUTH}(\mathcal{M})$ has two types of elements: authoritative $\bullet a$ and non-authoritative $\circ a$ (for $a \in \mathcal{M}$). For any instance γ , $\llbracket \bullet a \rrbracket^\gamma$ is *exclusive* (i.e., $\llbracket \bullet a \rrbracket^\gamma * \llbracket \bullet a \rrbracket^\gamma \vdash \top$), and is the main point of reference for all the non-authoritative fragments, in the sense that any ownable fragment $\llbracket \circ b \rrbracket^\gamma$ must have b *included* in a , that is $\exists c. a = b \cdot c$. The PCM's update therefore requires more: if one wants to update b to b' , it has not only to ensure b' is compatible with c , but also has to update a to $a' = b' \cdot c$. These properties are summarized in the following two rules:

$$\begin{array}{c} \text{Auth-Update} \\ \text{Auth-Agree} \quad \frac{\llbracket \bullet a \rrbracket^\gamma * \llbracket \circ b \rrbracket^\gamma \vdash \exists c. a = b \cdot c \quad \frac{b' \cdot c \neq \perp}{\llbracket \bullet b \cdot c \rrbracket^\gamma * \llbracket \circ b \rrbracket^\gamma \Rightarrow \llbracket \bullet b' \cdot c \rrbracket^\gamma * \llbracket \circ b \rrbracket^\gamma}}{\llbracket \bullet a \rrbracket^\gamma * \llbracket \circ b \rrbracket^\gamma \vdash \exists c. a = b \cdot c} \end{array}$$

With these two rules in hand, we can derive the following rules for operations on $\text{AUTH}(\text{HEAP})$:

$$\begin{array}{c} \text{AGhost-Heap-Exclusive} \quad \llbracket \circ [\ell := v] \rrbracket^\gamma * \llbracket \circ [\ell := w] \rrbracket^\gamma \vdash \top \quad \text{AGhost-Heap-Agree} \quad \llbracket \bullet \sigma \rrbracket^\gamma * \llbracket \circ [\ell := v] \rrbracket^\gamma \vdash \sigma(\ell) = v \\ \text{AGhost-Heap-Update} \quad \llbracket \bullet \sigma \rrbracket^\gamma * \llbracket \circ [\ell := w] \rrbracket^\gamma \Rightarrow \llbracket \bullet \sigma[\ell \mapsto v] \rrbracket^\gamma * \llbracket \circ [\ell := v] \rrbracket^\gamma \end{array}$$

We are now ready to establish the invariant $\llbracket \exists \sigma. \text{Phys}(\sigma) * \llbracket \bullet \sigma \rrbracket^\gamma \rrbracket$, which

[Jun+15] Jung et al., “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”

binds together the physical state ownership and the authoritative ghost heap ownership. With the invariant in place, **AGHOST-HEAP-AGREE** implies that if a thread owns the singleton ghost heap $\boxed{\circ}[\ell := v]$ locally, then, in combination with the invariant, it is guaranteed that ℓ currently has value v in the physical heap. **AGHOST-HEAP-EXCLUSIVE** and **AGHOST-HEAP-UPDATE** ensure that only the one thread who owns $\boxed{\circ}[\ell := v]$ can make updates to the contents of ℓ .

The points-to assertion is then defined as $\ell \hookrightarrow v \triangleq \boxed{\circ}[\ell := v]^\gamma$, and we can easily prove the small-footprint triples from the beginning of this section by combining the above rules for authoritative ghost heaps with those for opening and closing invariants.

4.1.2 Verifying MP in λ_{SC}

Using the small-footprint triples, we are ready to verify MP in λ_{SC} . We discuss the proof in a bit of detail here, since later on we will show how to adapt this proof to verify MP under weak-memory semantics.

The proof of MP is given in **Figure 4.3**. As a proof convention, we only mention persistent assertions (like invariants) once and use them freely later, since they are always true after being established. The proof works essentially as follows.

First of all, both threads want to operate on y simultaneously, so we need to put ownership of y into an invariant $\boxed{\text{Inv}_y}$. This invariant says that y is in one of two states—0 or 1. We can establish the invariant right after the initialization of y (the write of 0 to y), because y is in state 0 at that moment. The first thread is responsible for setting y to state 1. When the second thread observes that y is in state 1, it will expect to be able to gain ownership of $x \hookrightarrow 37$. To achieve this, in state 1, Inv_y asserts the existence of another invariant $\boxed{\text{Inv}_x}$ concerning x , and it is this latter invariant that we use to transfer ownership of location x from the first thread to the second thread.

To understand Inv_x , it helps to have seen the film *Raiders of the Lost Ark*, or at least the first few minutes of it, in which Indiana Jones (played by Harrison Ford) attempts to steal a precious golden idol from an ancient Peruvian temple—without setting off booby traps—by swapping it for a similarly weighted bag of sand. Unfortunately for him, the temple detects his ruse and tries to kill him. But we can play a similar trick, and Iris will be perfectly happy! In our case, the “golden idol” is $x \hookrightarrow 37$, which is transferred into the invariant $\boxed{\text{Inv}_x}$ when it is established by the first thread. The “bag of sand” is a “token” $\boxed{\diamond}$ (a uniquely ownable piece of ghost state) that is given to the second thread at the beginning of its execution. Inv_x simply asserts that it owns either the golden idol or the bag of sand. Thus, when the second thread learns of the existence of Inv_x , it can safely use the invariant opening and closing rules to swap the bag of sand in its possession for the golden idol owned by Inv_x , and thereafter claim local ownership of $x \hookrightarrow 37$.

INVARIANTS

$$\text{Inv}_y \triangleq y \hookrightarrow 0 \vee y \hookrightarrow 1 * \boxed{\text{Inv}_x}$$

$$\text{Inv}_x \triangleq [\diamond] \vee x \hookrightarrow 37$$

THREAD 1 PROOF OUTLINE

$$\begin{aligned} & \{x \hookrightarrow 0 * \boxed{\text{Inv}_y}\} \\ & x := 37 \\ & \{x \hookrightarrow 37\} \\ & \boxed{\text{Inv}_x} \\ & \text{open } \text{Inv}_y \left| \begin{array}{l} \{y \hookrightarrow _ \} \\ y := 1 \\ \{y \hookrightarrow 1 * \boxed{\text{Inv}_x}\} \end{array} \right. \\ & \{\perp\} \end{aligned}$$

THREAD 2 PROOF OUTLINE

$$\begin{aligned} & \{\boxed{\text{Inv}_y} * [\diamond]\} \\ & \text{repeat } y; \\ & \quad \{\boxed{\text{Inv}_x} * [\diamond]\} \\ & \quad \{x \hookrightarrow 37\} \\ & !x \\ & \{z. z = 37 * x \hookrightarrow 37\} \end{aligned}$$
Figure 4.3: Verification of MP in λ_{SC} .4.2 INSTANTIATING IRIS WITH λ_{RN}

We now consider an instantiation of Iris with our λ_{RN} language from §3.3. A key difference between λ_{RN} and λ_{SC} is that the expression reductions of λ_{SC} do not depend on which thread is executing the expression, since every thread has the same global view of the memory, whereas the reductions of λ_{RN} depend on the current thread's subjective view of the memory. Thus, we need to be able to talk about thread ids in our logic as well. To this end, we pair up expressions from λ_{RN} with thread ids, making them visible in our specifications. Eventually, in §5, we will see how we can reason about λ_{RN} without talking explicitly about thread ids.

4.2.1 Encoding Separation Logic for λ_{RN}

After instantiating Iris with λ_{RN} , as in the case of λ_{SC} , Iris provides us with large-footprint specifications of the primitive commands for free, which concern the physical state assertion and mirror the rules of λ_{RN} 's operational semantics. Recall that in λ_{RN} the physical state is a tuple (M, T, N) of the message pool M , the current view map T , and the non-atomic timestamp map N . As before, we aim to develop “local” assertions using ghost state, establish an invariant that connects those local assertions to the physical state assertion, and then derive small-footprint specifications of the primitive commands for use in modular verification. But what kind of “local” assertions do we want?

For λ_{SC} , we had the points-to assertion $\ell \hookrightarrow v$, but in λ_{RN} we no longer have a simple mapping from locations to values. Rather, associated with each location ℓ is a set of messages corresponding to writes to ℓ . We will represent that associated information as a *history* h , consisting of a set of triples (v, t, V) , where v is a value written to ℓ , t is the timestamp at which that value was written, and V is the view of the writing thread at the time the write occurred. Note that V incorporates the new timestamp, i.e., $V(\ell) = t$. To reflect the per-location history, we define our first local assertion: The *history ownership* assertion $\text{Hist}(\ell, h)$ asserts ownership of ℓ and knowledge of its write history h .

$$\begin{array}{l}
\text{Base-NA-Read} \\
\boxed{\text{PSInv}} \vdash \{ \text{Seen}(\pi, V) * \text{Hist}(\ell, h) * \text{init}(h, V) * \text{na}(h, V) \} \\
\ell_{[\text{na}]}, \pi \\
\{ v. \text{Seen}(\pi, V) * \text{Hist}(\ell, h) * \text{na}(h, V) * \max(h).v = v \} \\
\\
\text{Base-NA-Write} \\
\boxed{\text{PSInv}} \vdash \{ \text{Seen}(\pi, V) * \text{Hist}(\ell, h) * \text{alloc}(h, V) \} \\
\ell_{[\text{na}]} := v, \pi \\
\left\{ \begin{array}{l} \exists V' \sqsupseteq V, t, h' = \{(v, t, V')\}. \text{Seen}(\pi, V') * \text{Hist}(\ell, h') \\ * \text{na}(h', V') * \text{init}(h', V') \end{array} \right\} \\
\\
\text{Base-AT-Read} \\
\boxed{\text{PSInv}} \vdash \{ \text{Seen}(\pi, V) * \text{Hist}(\ell, h) * \text{init}(h, V) \} \\
\ell_{[\text{at}]}, \pi \\
\left\{ \begin{array}{l} v. \exists t_1, V_1, V' \sqsupseteq V \sqcup V_1. \text{Seen}(\pi, V') * \text{Hist}(\ell, h) \\ * (v, t_1, V_1) \in h * t_1 \geq V(\ell) \end{array} \right\} \\
\\
\text{Base-AT-Write} \\
\boxed{\text{PSInv}} \vdash \{ \text{Seen}(\pi, V) * \text{Hist}(\ell, h) * \text{alloc}(h, V) \} \\
\ell_{[\text{at}]} := v, \pi \\
\left\{ \begin{array}{l} \exists V' \sqsupseteq V, t, h' = h \uplus \{(v, t, V')\}. \text{Seen}(\pi, V') * \text{Hist}(\ell, h') \\ * \text{init}(h', V') \end{array} \right\}
\end{array}$$

Figure 4.5: Selected Hoare triples of λ_{RN} base logic.

these predicates require that V has seen *some* event from h —i.e., an event from the current era of ℓ —which, as discussed above, is a prerequisite for non-raciness.

These base logic rules provide a very concise explanation of λ_{RN} 's operational semantics. **BASE-AT-READ**, for example, requires the current view's knowledge of ℓ being initialized and ensures that the new updated view V' is at least the join of the local view V and the view V_1 of the write event that the thread reads from. This event must be from h and not *mo*-earlier than the write event observed by the thread previously. **BASE-NA-READ** is similar, except that it requires that the current view must have observed the *mo*-latest write event to ℓ , and therefore reads from that write event, which we denote by $\max(h)$. **BASE-NA-WRITE** preserves the HInv invariant by proactively dropping from the history all the old write events, which are *mo*-before this non-atomic write. Notice that, unlike **BASE-NA-READ**, **BASE-NA-WRITE** does not require $\text{na}(h, V)$, as explained in §3.2.

4.2.2 MP in λ_{RN}

We show that the base logic is enough to verify MP in λ_{RN} . The invariants and the proof, given in Figure 4.6, follow closely those used for MP in λ_{SC} . The singleton heap ownership in λ_{SC} is replaced with the history ownership, and extra conditions on view extension are added to reflect the view updates inherent in λ_{RN} . More specifically, in Inv_y , we enforce

Figure 4.6: Verification of MP in λ_{RN} base logic.

INVARIANTS

$$\begin{aligned}
\text{Inv}_y(V_0) &\triangleq \exists h. \text{Hist}(y, h) * (0, _, V_0) \in h \\
&\quad * \forall V_1, v_1 \neq 0. (v_1, _, V_1) \in h \Rightarrow \exists V_{37} \sqsubseteq V_1. \boxed{\text{Inv}_x(V_{37})} \\
\text{Inv}_x(V_{37}) &\triangleq \boxed{\diamond} \vee \text{Hist}(x, [(37, _, V_{37})])
\end{aligned}$$

THREAD 1 PROOF OUTLINE

$$\begin{aligned}
&\{ \text{Seen}(\pi, V_0) * \text{Hist}(x, [(0, _, V_x)]) * V_x \sqsubseteq V_0 * \boxed{\text{Inv}_y(V_0)} \} \\
&x_{[\text{na}]} := 37 \\
&\{ \exists V_{37} \sqsupseteq V_0. \text{Seen}(\pi, V_{37}) * \text{Hist}(x, [(37, _, V_{37})]) \} \\
&\{ \text{Seen}(\pi, V_{37}) * \boxed{\text{Inv}_x(V_{37})} \} \\
&\text{open } \text{Inv}_y \left| \begin{array}{l} \{ \text{Seen}(\pi, V_{37}) * \exists h. \text{Hist}(y, h) * \dots \} \\ y_{[\text{at}]} := 1 \\ \{ \exists V_1 \sqsupseteq V_{37}. \text{Seen}(\pi, V_1) * \text{Hist}(y, h \uplus [(1, _, V_1)]) * \boxed{\text{Inv}_x(V_{37})} \} \end{array} \right. \\
&\{ \text{Seen}(\pi, V_1) * \boxed{\text{Inv}_y(V_0)} \}
\end{aligned}$$

THREAD 2 PROOF OUTLINE

$$\begin{aligned}
&\{ \text{Seen}(\pi, V_0) * \boxed{\text{Inv}_y(V_0)} * \boxed{\diamond} \} \\
&\text{repeat } y_{[\text{at}]}; \\
&\{ \exists V_1, V_{37}, V_2. V_2 \sqsupseteq V_1 \sqsupseteq V_{37} * \text{Seen}(\pi, V_2) * \boxed{\text{Inv}_x(V_{37})} * \boxed{\diamond} \} \\
&\{ \text{Seen}(\pi, V_2) * V_{37} \sqsubseteq V_2 * \text{Hist}(x, [(37, _, V_{37})]) \} \\
&x_{[\text{na}]} \\
&\{ z. \text{Seen}(\pi, V_2) * z = 37 * \text{Hist}(x, [(37, _, V_{37})]) \}
\end{aligned}$$

⁴ In the MP example this value is always 1.

that any write of a non-zero value⁴ to y be at a view V_1 that extends V_{37} , which is the view at which the write of 37 to x is made by the first thread. Consequently, when the second thread observes V_1 (by reading y to be non-zero), it must have also observed V_{37} , and thus can read $x = 37$, using the Indiana Jones invariant Inv_x . The extra conditions on V_0 ensure that y is initialized with 0 at V_0 , so that the second thread (having observed V_0) can safely read y .

We have shown that the base logic is powerful enough to verify MP in λ_{RN} , and in principle it is powerful enough to verify many other realistic weak memory programs that are expressible in λ_{RN} as well. However, it is also clear that the base logic is not abstract enough: one has to burden oneself with keeping track of the low-level details of changes to locations' histories and threads' views. What we really want is a way of abstracting away from those low-level details and finding simple high-level reasoning principles for λ_{RN} , the type of reasoning principles supported by RA+NA logics like GPS and RSL. We will now see how such high-level principles can in fact be derived on top of our low-level base logic.

5

iGPS

Vafeiadis *et al.* have introduced several logics for C11, and two in particular that were focused on RA+NA: GPS [TVD14] and RSL [VN13]. The key difference between these logics and the RA+NA base logic presented in §4.2.1 is that, in GPS and RSL, the user does not reason explicitly about views—instead, the assertions of these logics are *implicitly* predicated over the view of the thread asserting them. This helps to hide much tedious reasoning about views, and leads naturally to a model of assertions as predicates over views (§5.3).

We have encoded iGPS and iRSL, variants of both GPS and RSL, in Iris, and we will focus here on iGPS, since it is the more sophisticated of the two logics. (We briefly describe iRSL in §6.) GPS introduced several useful abstractions that were not present in RSL: PCM-based ghost state, single-location protocols, and escrows. In encoding GPS in Iris, as noted in the introduction, we get PCM-based ghost state completely for free, just by working in Iris. Below, we will describe the other features, and how we account for them in Iris.

Note that our goal with iGPS is not to slavishly imitate all details of GPS, but to provide proof rules very similar to GPS’s that are both strong enough to support all the examples from the GPS papers [TVD14; TDV15] and significantly easier to prove sound within Iris. To this end, we slightly restrict select rules, but only in a way that does not impact their utility on known case studies. We discuss the differences from the original rules in detail in §7. Furthermore, in §5.2, we show how iGPS supports an additional feature—*single-writer protocols*—that was not supported by the original GPS and that significantly simplifies proofs.

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A program logic for C11 concurrency”

[TDV15] Tassarotti et al., “Verifying Read-copy-update in a Logic for Weak Memory”

5.1 KEY FEATURES OF GPS

In this section, we explain the key features of GPS and how they are formalized in iGPS (besides PCM-based ghost state, which is directly imported into iGPS from Iris). A selected set of iGPS proof rules is given in Figure 5.1.

Figure 5.1: iGPS proof rules for non-atomics, protocols, and escrows.

$$\begin{array}{c}
\text{iGPS-NA-Read} \quad \{\ell \hookrightarrow v\} \ell_{[\text{na}]} \{w. w = v * \ell \hookrightarrow v\} \quad \text{iGPS-NA-Write} \quad \{\ell \hookrightarrow _ \} \ell_{[\text{na}]} := v \{ \ell \hookrightarrow v \} \\
\\
\text{iGPS-NA-Exclusive} \quad \ell \hookrightarrow v * \ell \hookrightarrow w \Rightarrow \perp \\
\\
\text{iGPS-Read} \quad \frac{\forall s' \sqsubseteq s, v. P * \tau_{\text{read}}(s', v) \Rightarrow Q}{\{\ell : s \mid \tau * P\} \ell_{[\text{at}]} \{v. \exists s' \sqsubseteq s. \ell : s' \mid \tau * Q\}} \\
\\
\text{iGPS-Write} \quad \frac{(\forall s'', s' \sqsubseteq s'') \quad P \Rightarrow \tau_{\text{full}}(s', v) * Q}{\{\ell : s \mid \tau * P\} \ell_{[\text{at}]} := v \{ \ell : s' \mid \tau * Q \}} \\
\\
\text{iGPS-CAS} \quad \frac{\forall s' \sqsubseteq s. P * \tau_{\text{full}}(s', v_o) \Rightarrow \exists s'' \sqsubseteq s'. \tau_{\text{full}}(s'', v_n) * Q \quad \forall s' \sqsubseteq s. P * \tau_{\text{read}}(s', v_o) \Rightarrow R}{\left\{ \ell : s \mid \tau * P \right\} \text{cas}(\ell, v_o, v_n) \left\{ v. \exists s'' \sqsubseteq s. \ell : s'' \mid \tau \right.} \\
\left. * ((v = 1 \wedge Q) \vee (v = 0 \wedge R)) \right\} \\
\\
\text{iGPS-Persistent} \quad \ell : s \mid \tau \Rightarrow \Box \ell : s \mid \tau \quad \text{iGPS-Agree} \quad \ell : s_1 \mid \tau * \ell : s_2 \mid \tau \Rightarrow s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1 \\
\\
\text{iGPS-Escrow-Intro} \quad Q \Rightarrow [P \rightsquigarrow Q] \quad \text{iGPS-Escrow-Elim} \quad P \wedge [P \rightsquigarrow Q] \Rightarrow Q \quad \text{iGPS-Escrow-Persistent} \quad [P \rightsquigarrow Q] \Rightarrow \Box [P \rightsquigarrow Q]
\end{array}$$

Non-atomics

Since non-atomic locations may not be raced on, GPS (and iGPS) reason about them much in the same way that locations are reasoned about in standard separation logic: using the points-to assertion, $\ell \hookrightarrow v$. Note that the proof rules for reading (**iGPS-NA-READ**) and writing (**iGPS-NA-WRITE**) and the exclusivity property (**iGPS-NA-EXCLUSIVE**) are equivalent to those from the logic for λ_{SC} (§4.1.1). Additionally, GPS (and iGPS) support fractional ownership of non-atomics to allow such locations to be read (but not written) by multiple threads at once. We omit the rules of fractional ownership for brevity.

Protocols

To reason about RA atomics, we need a mechanism for controlling interference on such accesses. Toward this end, CSLs for SC have supported a variety of *protocol* mechanisms, which control how shared state may evolve over time, and several of the more recent logics [TDB13; SB14; Nan+14] employ *state transition systems* (STSs) to formalize such protocols. Crucially, protocols enforce *irreversibility*: the state of an STS protocol can only make forward progress over the course of a proof. For example, in §4.1.2, a protocol could enforce that the variable y could only progress from 0 to 1 but not back again. (We did not need to enforce that property to verify the MP example, but it is useful to be able to in

[TDB13] Turon et al., “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”

[SB14] Svendsen et al., “Impredicative Concurrent Abstract Predicates”

[Nan+14] Nanevski et al., “Communicating State Transition Systems for Fine-Grained Concurrent Resources”

general.) In Iris, protocols are encoded using a combination of invariants and ghost state.

Under weak memory, invariants and protocols are unsound in general because they require a single coherent history of updates to all locations. GPS showed how to partially restore protocol reasoning for weak memory with *single-location protocols*: protocols which restrict the evolution of a single shared location. Intuitively, single-location protocols are sound due to the per-location coherence property of C11 (often called “SC per location”): the writes to any single location are totally ordered (by *mo*). In particular, they maintain the invariant that the order of protocol states associated with writes is consistent with their timestamp (*mo*) order. If write event x to location ℓ with associated protocol state s_x is *mo*-before write event y (to the same location) with protocol state s_y , then s_x will be before s_y in protocol order. Thus, once a thread has observed that the protocol on ℓ has reached state s_x , it can from that point on only observe the protocol on ℓ to be in states that are accessible from s_x . This fulfills the expectation that protocol transitions are irreversible.

GPS protocols come equipped with an interpretation predicate which specifies the resources held by the protocol depending on the protocol’s state and the location’s value. The primitive operations on an atomic location serve to transfer resources in and out of its protocol:

- Writes may transfer resources into the protocol, but may not transfer anything out.
- Reads may not transfer any resources into the protocol, but they may transfer “knowledge” (*i.e.*, duplicable resources) out of it. They are restricted to transferring out duplicable resources because there may be many reads of the same write event.
- Updates (RMWs), by virtue of the physical synchronization they provide, may transfer resources both in and out of the protocol.

In iGPS, we represent protocols in a slightly different way, using *two* interpretations: a “full” interpretation, and a duplicable “read” interpretation that is implied by the full interpretation. The intuition is that the read interpretation is used for reads (since they may only transfer duplicable resources out of the protocol) and the full interpretation is used for the other operations.

We will now present the formal definition of protocols and the corresponding proof rules.

Definition 5.1.1 (Protocols). *A protocol τ comprises a non-empty state set $\mathbb{S}$, a reflexive, transitive transition relation $\sqsubseteq \subseteq \mathbb{S} \times \mathbb{S}$, and two interpretation predicates $\tau_m(\cdot, \cdot) \in \mathbb{S} \times \text{Val} \rightarrow \text{Prop}$ with $m \in \{\text{read}, \text{full}\}$ representing read and full interpretations, respectively. The interpretation predicate has to fulfill the following two laws:*

$$\tau_{\text{full}}(s, v) \Rightarrow \tau_{\text{full}}(s, v) * \tau_{\text{read}}(s, v) \quad \tau_{\text{read}}(s, v) \Rightarrow \tau_{\text{read}}(s, v) * \tau_{\text{read}}(s, v)$$

We write $\boxed{\ell : s \mid \tau}$ (as in GPS) to denote the persistent assertion that ℓ is governed by protocol τ and that the protocol has been observed in state s .

The first rule, **iGPS-AGREE**, represents the guarantee that every protocol is always in a state consistent with all observations, *i.e.*, that all observed states can be linearly ordered w.r.t. to the transition relation $\sqsubseteq$.

iGPS-READ enables reading from a location governed by protocol τ —allowing the user to observe a future state s' of whatever state s it has previously observed, and providing the associated read interpretation τ_{read} .

Writes to the location are subject to **iGPS-WRITE**, which allows the user to move the protocol to a “final state” s' —*i.e.*, a state accessible from every state in the protocol—so long as they can provide τ_{full} for s' . This rule may seem very weak, since it forces the protocol into a final state, but this weakness derives from the need to handle the general case where there can be write-write races. Write-write races allow only very limited reasoning: both writes have to prove that their protocol state is later in protocol order to the other one. The write rule presented here solves this problem in a very simple way (see §7 for a comparison with the original GPS rule). In §5.2, we present a much stronger write rule that is optimized for the common case where there are no write-write races on the location.

Finally, **iGPS-CAS** governs updates. Its two premises represent the success and failure case, respectively. If the operation succeeds, the value read, v_o , is guaranteed to belong to a future state s' . The user picks the new state s'' depending on s' and establishes $\tau_{\text{full}}(s'', v_n)$, making use of $\tau_{\text{full}}(s', v_o)$. In case of a failure, the rule degenerates to **iGPS-READ**.

Escrows

A limitation of GPS protocols is that they offer no way to transfer ownership of (non-duplicable) resources from one thread to another unless the receiving thread performs physical synchronization via an update operation. For example, in our MP example, there is no update operation, and yet we want to transfer ownership of the non-atomic location x from the first thread to the second. For such an example, an additional mechanism for ownership transfer is required.

This motivates *escrows*, a mechanism for *logical* synchronization which, unlike protocols, is not tied to physical locations.

Definition 5.1.2 (Escrows). *An escrow $[P \rightsquigarrow Q]$ consists of a guard resource P and a payload resource Q to be transferred. The guard resource P must be exclusive, *i.e.* $P * P \Rightarrow \perp$. The escrow assertion itself is persistent knowledge (freely duplicable).*

The idea of escrows is really just a slight generalization of the “Indiana Jones invariant” Inv_x that we used in the proof of the MP example from §4.1.2. Following the explanation there, the payload resource Q is the “golden idol”, the guard resource P is the “bag of sand”, and the escrow

allows one to swap P for Q . The restriction on exclusivity of P ensures that this swap can only be performed once.

The proof rules for escrows follow the above intuition. **IGPS-ESCROW-INTRO** places the payload resource Q in escrow. Any thread that learns of the existence of this escrow can then use **IGPS-ESCROW-ELIM** to trade ownership of the guard resource P for Q .¹

Message passing in iGPS

The verification of MP using iGPS is given in **Figure 5.2**. Although the verification is sound for λ_{RN} , it is much simpler than the proof we carried out in the base logic of Iris, and is in fact very close in structure to the SC verification of MP in λ_{SC} . In particular, the Indiana Jones invariant Inv_x has now become an escrow **XE**, and the invariant Inv_y has now become a (2-state) iGPS protocol **YP**, but otherwise the steps are almost the same. The abstraction of iGPS has relieved us from the burden of reasoning with history and view updates explicitly.

$$\begin{aligned}
 \mathbf{XE}(x) &\triangleq \left[\begin{array}{c} \boxed{} \\ \boxed{} \end{array} \rightsquigarrow x \hookrightarrow 37 \right] \\
 \mathbf{YP}(x)(0, v) &\triangleq v = 0 \\
 \mathbf{YP}(x)(1, v) &\triangleq v = 1 * \mathbf{XE}(x)
 \end{aligned}$$

$$\left\{ \begin{array}{l} x \hookrightarrow 0 * \boxed{y : 0 \mid \mathbf{YP}(x)} \\ x_{[\text{na}]} := 37 \\ \{x \hookrightarrow 37\} \\ \{\mathbf{XE}(x) * \boxed{y : 0 \mid \mathbf{YP}(x)}\} \\ y_{[\text{at}]} := 1 \\ \boxed{y : 1 \mid \mathbf{YP}(x)} \end{array} \right\} \parallel \left\{ \begin{array}{l} \boxed{y : 0 \mid \mathbf{YP}(x)} * \boxed{} \\ \text{repeat } y_{[\text{at}]}; \\ \boxed{y : 1 \mid \mathbf{YP}(x)} * \mathbf{XE}(x) * \boxed{} \\ \{x \hookrightarrow 37\} \\ x_{[\text{na}]} \\ \{z. z = 37 * x \hookrightarrow 37\} \end{array} \right\}$$

Figure 5.2: Verification of MP with iGPS

5.2 SINGLE-WRITER PROTOCOLS

As we observed above, the iGPS protocol write rule suffers from a restriction: the user has to transition to a final state. This restriction is not present in CSLs for SC, which let the user pick the future state depending on the current one, much as **IGPS-CAS** does. Fortunately, in the common case when there are no write-write races to the location, this restriction can be lifted by *single-writer protocols*, a novel invention of iGPS.

A single-writer protocol splits the protocol assertion into two parts: an exclusive writer assertion $\boxed{\ell : s \mid \tau}_W$ and a persistent reader assertion $\boxed{\ell : s \mid \tau}_R$. Owning the writer assertion provides both the permission to change the state as well as the guarantee that no one else can change it. Owning the reader assertion only allows reads. Thus, the reader assertion represents a lower bound on the protocol state whereas the state contained in the writer assertion is exactly the most recent state of the protocol. The full proof rules for single-writer protocols are given in **Figure 5.3**, and of these, the write rule **IGPS-SW-WRITE** is the most

¹ The rule given for elimination is only sound if Q is “timeless”, meaning that it only describes ownership of (ghost) state, not knowledge about protocols or escrows, as is the case in our message passing example. A more general rule, which returns Q under the later modality, is given in **Appendix B.1**.

important. The writer knows exactly what the current state is, and is free to pick the next state accordingly.

Figure 5.3: A selection of single-writer proof rules.

$$\begin{array}{c}
\text{iGPS-SW-Exclusive-Writer} \\
\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_W \Rightarrow \perp \\
\\
\text{iGPS-SW-Agree} \\
\boxed{\ell : s_1 \mid \tau}_R * \boxed{\ell : s_2 \mid \tau}_R \Rightarrow s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1 \\
\\
\text{iGPS-SW-Max} \\
\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_R \Rightarrow s_1 \sqsupseteq s_2 \\
\\
\text{iGPS-SW-Read-Exclusive} \\
\left\{ \boxed{\ell : s \mid \tau}_W \right\} \ell_{[\text{at}]} \left\{ v. \boxed{\ell : s \mid \tau}_W * \tau_{\text{read}}(s, v) \right\} \\
\\
\text{iGPS-SW-Read} \\
\frac{\forall s' \sqsupseteq s, v. P * \tau_{\text{read}}(s', v) \Rightarrow Q}{\left\{ \boxed{\ell : s \mid \tau}_R * P \right\} \ell_{[\text{at}]} \left\{ v. \exists s' \sqsupseteq s. \boxed{\ell : s' \mid \tau}_R * Q \right\}} \\
\\
\text{iGPS-SW-Write} \\
\frac{P * \boxed{\ell : s'' \mid \tau}_W * \tau_{\text{full}}(s, -) \Rightarrow \tau_{\text{full}}(s'', v) * Q \quad s'' \sqsupseteq s}{\left\{ \boxed{\ell : s \mid \tau}_W * P \right\} \ell_{[\text{at}]} := v \left\{ \boxed{\ell : s'' \mid \tau}_R * Q \right\}}
\end{array}$$

Applications of single-writer protocols

Single-writer protocols provide more explicitly intuitive and concise proofs over normal protocols when there are no *write-write* races, *i.e.*, only one thread is writing to the location. This may mean that there is exactly one writer in the whole program, or (perhaps more typically) that the programmer is using sufficient synchronization to ensure that there is exactly one active writer at a time. Such is the case for several headlining examples verified in GPS, including *circular buffer*, *bounded ticket lock*, and *read-copy update* [TVD14; TDV15].

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”

[TDV15] Tassarotti et al., “Verifying Read-copy-update in a Logic for Weak Memory”

In the original GPS proofs for these examples, the lack of single-writer protocols meant that the proofs had to employ a significant amount of tedious ghost state (mostly in the form of so-called “protocol tokens”) to formalize the fact that the writing thread knew exactly which state the protocol had to be in at any given time. Using single-writer protocols, this reasoning is immediate from the **iGPS-SW-WRITE** rule. By removing the need for such boilerplate ghost state, single-writer protocols simplify and clarify the proofs of these examples.

An intriguing feature of the **iGPS-SW-WRITE** rule is that it is possible for the writer to relinquish ownership of the exclusive writer permission *while doing the write itself*. (This is why the writer permission appears in the precondition of the premise.) This extra flexibility is particularly useful when reasoning about RA implementations of locks (such as the bounded ticket lock). When the lock holder releases the lock (typically with a release write), this feature allows them to also give up their permission to do further release writes to the lock, so that it can be transferred to the

next thread that acquires the lock.

5.3 THE MODEL OF IGPS

$$\begin{aligned}
\llbracket \bar{a} \rrbracket^\gamma &\triangleq \lambda _ . \llbracket \bar{a} \rrbracket^\gamma \\
\llbracket \Box P \rrbracket &\triangleq \lambda V . \Box \llbracket P \rrbracket(V) \\
\llbracket P * Q \rrbracket &\triangleq \lambda V . \llbracket P \rrbracket(V) * \llbracket Q \rrbracket(V) \\
\llbracket P \Rightarrow Q \rrbracket &\triangleq \lambda V . \forall V' \supseteq V . \llbracket P \rrbracket(V') \Rightarrow \llbracket Q \rrbracket(V') \\
\llbracket \ell \hookrightarrow v \rrbracket &\triangleq \lambda V . \exists V_{\text{na}} \sqsubseteq V . \text{Hist}(\ell, \{(v, _, V_{\text{na}})\}) \\
\llbracket [P \rightsquigarrow Q] \rrbracket &\triangleq \lambda V . \exists V_0 \sqsubseteq V . \llbracket P \rrbracket(V_0) \vee \llbracket Q \rrbracket(V_0) \\
\llbracket \{P\} e \{v. Q\} \rrbracket &\triangleq \lambda V . \forall V' \supseteq V, \pi . \\
&\quad \{ \boxed{\text{PSInv}} * \text{Seen}(\pi, V') * \llbracket P \rrbracket(V') \} \\
&\quad (e, \pi) \\
&\quad \{ v. \exists V'' \supseteq V' . \text{Seen}(\pi, V'') * \llbracket Q \rrbracket(V'') \}
\end{aligned}$$

Figure 5.4: Definition of iGPS assertions.

We now briefly describe the model of iGPS assertions. [Figure 5.4](#) contains an excerpt of the encoding of the standard assertions and connectives of CSL as well as non-atomics and escrows. The somewhat more involved model of protocols is detailed in [Appendix C](#).

We model iGPS assertions as monotone predicates over views. The view parameter represents the current view of the thread making the assertion. The monotonicity requirement is motivated by the observation that the view of a thread only grows over the execution of a program. To ensure properties like the frame rule, it is therefore crucial that simply adding information to a view does not invalidate previously held (e.g., framed) assertions. As a consequence of this requirement, we explicitly monotone the encoding when necessary.

We benefit greatly from the support offered by the surrounding logic. As a result, the model is extremely simple, with the lion's share of connectives being translated in a purely structural way and the remaining ones making direct use of ambient Iris connectives. The most interesting encodings are those of Hoare triples, non-atomics, escrows, and protocols. We now discuss these in more detail.

The model of Hoare triples embodies the intuition behind our encoding of iGPS assertions as view predicates: the view at which we operate is that of the local thread. In the encoding, we achieve this by quantifying over a view V' and tying V' to the physical view of the thread π via the $\text{Seen}(\pi, V')$ assertion and to the original pre-condition P , which is required to hold at V' . As the thread's physical view may evolve during the execution of the expression e , the triple returns an extended view $V'' \supseteq V'$ and the corresponding $\text{Seen}(\pi, V'')$ assertion, together with the post-condition Q , which is guaranteed to be valid at V'' .

The encoding of non-atomics is particularly simple due to the properties of the Hist assertion. As all writes in the history have to be *mo*-after the most recent non-atomic write, the history becomes a singleton when the location is used non-atomically. To tie the local view V to the view of the non-atomic write V_{na} , we simply demand that V extend V_{na} .

We encode escrows with a single, simple invariant, which holds either the guard resource P or the payload resource Q . The view V_0 at which the invariant owns one of these resources is the view used to initialize the escrow. Knowing an escrow at a local view V thus reduces to knowing that $V_0 \sqsubseteq V$.

Protocols

The model of iGPS protocols consists of two parts: a protocol invariant, and local protocol assertions given out to clients. The protocol invariant owns the location's history as well as the *logical history* Δ , which tracks the transition history of protocol states and is always kept in agreement with the location's history. Additionally, the invariant owns τ_{read} for all writes in the history and τ_{full} for select ones, depending on the protocol's type. For example, in normal protocols, τ_{full} is only stored for CAS-able write events, justifying that only an update can access the full interpretation of the write event that it reads from (see **iGPS-CAS**). Meanwhile, local protocol assertions hold knowledge about the logical history, which gives effectively lower bounds on the current state of the protocol, and by the protocol invariant, indirectly implies knowledge about the location's history. In the case of single-writer protocols, the writer assertion also owns the exclusive right to change the state. This construction also relies on the authoritative PCM (see §4.1.1). More details are given in **Appendix C**.

Soundness of iGPS

The soundness of iGPS is expressed by the following theorem.

Theorem I.1 (Adequacy). *For any expression e , physical state σ' , and meta-level predicate on values Φ , we have*

$$\begin{aligned} (\vdash \{\top\} e \{v. \Phi(v)\}) &\Rightarrow \forall \mathcal{TS}. \sigma_{\text{init}}; [0 \mapsto e] \rightarrow^* \sigma'; \mathcal{TS} \Rightarrow \\ &(\mathcal{TS}(0) \in \text{Val} \Rightarrow \Phi(\mathcal{TS}(0))) \\ &\wedge \forall \rho \in \text{dom}(\mathcal{TS}). \mathcal{TS}(\rho) \in \text{Val} \vee \exists \sigma'', e'', e_f''. \mathcal{TS}(\rho) \rightarrow^\rho \sigma'', e'', e_f'' \end{aligned}$$

The theorem connects iGPS program specifications $(\vdash \{\top\} e \{v. \Phi(v)\})$ and the program's possible executions, and provides two guarantees:

1. If the original thread with id 0 terminates with a value v , we know that $\Phi(v)$ holds.
2. For any pair of a state σ' and a threadpool $\mathcal{TS}$ reachable from the initial state σ_{init} (see **Definition 3.2.1**) and the initial threadpool $[0 \mapsto e]$, we know that any thread ρ in $\mathcal{TS}$ either has terminated with a value or can still be reduced in the state σ' .

The proof follows from the adequacy theorem of Iris.

6

Other Contributions

In our Rocq development accompanying this chapter, we make several additional contributions that we briefly summarize here.

An RSL encoding

Using the same base logic from §4.2.1, we have mechanized iRSL, a higher-order variant of RSL [VN13]. RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are $\text{Rel}(\ell, Q)$ and $\text{Acq}(\ell, Q)$, representing the permission to write to and read from ℓ , respectively. The resource $Q(v)$ is released by writing v to ℓ , and then acquired by reading v from ℓ . Consequently, to support this MP-like mechanism, the encoding’s model shares a great deal with the model of iGPS, namely the full vs. read interpretation construction for protocols.

One particular feature of RSL, however, demands special attention. Although simpler than GPS, RSL has the extra ability to split the receiver predicate Q into *smaller* predicates, so that different acquire reads of the same value v can acquire different parts of the transferred resource: $\text{Acq}(\ell, \lambda v. Q_1(v) * Q_2(v)) \Rightarrow \text{Acq}(\ell, Q_1) * \text{Acq}(\ell, Q_2)$. It is not obvious how to prove this sound when the splitting is completely arbitrary. Fortunately, a similar pattern, called the *barrier* pattern, has been addressed by Jung *et al.* [Jun+16], who propose the mechanism of “higher-order ghost state” to support such splitting. Our iRSL model basically extends Jung *et al.*’s barrier proof with a more complex (Iris) protocol to carefully manage resource splitting.

The encoding in Iris also provides us with useful extensions to the logic. Without extra work, iRSL naturally supports PCM-based ghost state and higher-order assertions, which were not available in the original RSL. The encoding shows that our approach has the right, reusable foundations to construct different logics for RA+NA.

In our RSL encoding, assertions are encoded as view predicates and proof rules are proven sound with respect to the base logic—in the same way as our GPS encoding. This allows us to soundly combine RSL and GPS reasoning principles in the same proof *at no additional cost*. It is even possible to design iGPS protocols whose state interpretation mentions iRSL assertions and vice versa. Of course, at a single point in time a

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A program logic for C11 concurrency”

[Jun+16] Jung et al., “Higher-order ghost state”

location can only be governed by either iGPS or iRSL, as they represent incompatible modes of ownership transfer.

Allocation and deallocation

We have also incorporated support for memory allocation and deallocation into our RA+NA operational semantics. Since C11 is not clear about the semantics of allocation and deallocation, we take the liberty of defining them as reasonably as possible: in short, allocation and deallocation behave as non-atomic writes with special values A and D , respectively.

Fractional protocols

So far, all protocols presented are permanent: once the protocols are established, they govern their locations forever. This poses two interesting questions: 1) Can we change the protocol which governs a location? and 2) How can we deallocate a location governed by a protocol? To support these features, we derive, with little modification to the iGPS model, *fractional protocols*, whose protocol assertions also assert the permission to even *use* the protocol. Initially, a protocol τ for a location ℓ will be established with the full fraction, and then it will be distributed to those who want to use τ . Later, when the full fraction is recollected, one can *disable* the protocol (since no one else can use it), regain the *raw* ownership of ℓ , and then deallocate ℓ —or more interestingly, establish a new protocol for ℓ ! These *recollectable* protocols open up possibilities for verifying programs that do custom memory reclamation, e.g., RCU (see below). In the current Rocq development, we have created fractional versions of both normal and single-writer protocols.

Mechanization and Case Studies

[KTB17] Krebbers et al., “Interactive proofs in higher-order concurrent separation logic”

Our Rocq mechanization employs a shallow embedding of iGPS (and iRSL), making critical use of the Iris Proof Mode [KTB17]. In its current form, the proof mode is specific to the algebra of Iris and offers no additional support for embedded logics like our encoding of iGPS assertions. There are two consequences to this: 1) Unlike in the paper presentation of iGPS, where thread views are completely hidden in the (Iris) model of the logic, in our Rocq proofs thread views are visible. However, they are also unobtrusive: all assertions in a given proof context always hold at the current thread’s local view, and the view only changes when the thread takes a step. Thus, while the views are visible, they are always manipulated and kept in sync in a very straightforward way, which we mostly automate with a set of simple tactics. 2) iGPS assertions cannot always be manipulated directly by the proof mode. We sometimes have to unfold our embedding of iGPS assertions—but not in the statements of our lemmas and theorems—to make explicit the underlying Iris assertions so that the proof mode can operate on them. As our embedding is very simple, this has little additional overhead. In particular, all lemmas and theorems stated at the iGPS level remain easily applicable even to the

unfolded definitions at the Iris level.

We have verified all of the standard examples that have been proven in previous work. The simplest of these is the *spin-lock* example, proven in iRSL. More interestingly, using iGPS, we have also mechanized the *message passing*, *circular buffer*, *bounded ticket lock*, and *Michael-Scott lock-free queue* examples, which were only verified by hand in the original GPS paper. We have also verified a variant of the *read-copy update* (RCU) technique employed in the Linux kernel, following the proof of Tassarotti *et al.* [TDV15]. The RCU proof is the most substantial example in iGPS, which simplifies the original proof in GPS by using fractional single-writer protocols that allow garbage collection. To our knowledge, our development provides the very first mechanized proofs of the circular buffer, bounded ticket lock, Michael-Scott queue, and RCU examples in a weak-memory setting.

[TDV15] Tassarotti et al., “Verifying Read-copy-update in a Logic for Weak Memory”

7

Related Work

We demonstrate one of the first major applications of the Iris framework. Other recent applications include Krebbers *et al.* [KTB17], who developed the interactive Rocq proof mode for Iris that we rely on heavily, and Krogh-Jespersen *et al.* [KSB17], who use Iris to encode a logical-relations model of a relational model of a type-and-effect system for a higher-order, concurrent programming language. Neither of those papers considers weak memory.

There are a number of program logics for weak memory models [Sie+15; BAP15; AM16; LV15], some of which have mechanized soundness proofs [VN13; DV16; Rid10], but none of which provide real support for mechanized proofs of weak-memory programs in the way that we do.

FSL++ [DV17], an extension of FSL [DV16] (with ghost state) and RSL (with relaxed accesses), was used to mechanize a proof of an implementation of atomic reference counters based on the one in Rust’s Arc library. The proof is done by applying the basic laws of separation logic, resulting in painful manual work. Our approach alleviates a great deal of such tedium using the Iris proof mode. As of now, however, iGPS cannot reason about relaxed accesses.

RSL, FSL, and FSL++ have been encoded in Viper [MSS16] to provide an automated verification approach to weak memory programs [Sum17]. The encodings, however, axiomatize all proof rules without providing soundness guarantees, and are specific to the style of RSL and its FSL descendants. Our base logic, in contrast, is not tied to any specific surface logic and can be used to develop and prove sound different surface logics. It remains to be seen if the more expressive GPS protocols can be encoded in Viper.

iGPS is based on the GPS logic [TVD14] and supports all reasoning mechanisms of GPS. However, the exact rules of iGPS differ in various small ways from the original ones in GPS. These differences stem from pragmatic choices made to simplify the soundness proof of iGPS. A particularly noteworthy difference from GPS appears in the premises of the **iGPS-READ** and **iGPS-WRITE** rules: the split of the protocol interpretations τ into τ_{read} and τ_{full} . We show the original GPS rules below for

[KTB17] Krebbers et al., “Interactive proofs in higher-order concurrent separation logic”

[KSB17] Krogh-Jespersen et al., “A relational model of types-and-effects in higher-order concurrent separation logic”

[Sie+15] Sieczkowski et al., “A Separation Logic for Fictional Sequential Consistency”

[BAP15] Bornat et al., “New Lace and Arsenic: adventures in weak memory with a program logic”

[AM16] Abe et al., “Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models”

[LV15] Lahav et al., “Owicki-Gries Reasoning for Weak Memory Models”

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A program logic for C11 concurrency”

[DV16] Doko et al., “A Program Logic for C11 Memory Fences”

[Rid10] Ridge, “A rely-guarantee proof system for x86-TSO”

[DV17] Doko et al., “Tackling Real-Life Relaxed Concurrency with FSL++”

[MSS16] Müller et al., “Viper: A Verification Infrastructure for Permission-Based Reasoning”

[Sum17] Summers

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”

comparison.

$$\begin{array}{c}
 \text{GPS-Read} \\
 \frac{\forall s' \sqsupseteq s. P * \tau(s', v) \Rightarrow \Box Q}{\left\{ \boxed{\ell : s \mid \tau} * P \right\} \ell_{[\text{at}]} \left\{ v. \boxed{\ell : s' \mid \tau} * Q \right\}} \\
 \\
 \text{GPS-Write} \\
 \frac{\forall s' \sqsupseteq s. P * \tau(s', v) \Rightarrow s' \sqsubseteq s'' \quad P \Rightarrow \tau(s'', w) * Q}{\left\{ \boxed{\ell : s \mid \tau} * P \right\} \ell_{[\text{at}]} := w \left\{ \boxed{\ell : s'' \mid \tau} * Q \right\}}
 \end{array}$$

The interpretation $\tau(s, v)$ in these two GPS rules is equivalent to $\tau_{\text{full}}(s, v)$ in iGPS. In GPS, the user can gain access to the full interpretation of the “current” protocol state (s'), but only to obtain some *knowledge*, not to consume (i.e., transfer out of the protocol) any non-duplicable resources owned by that interpretation. This is enforced in **GPS-READ** by guarding Q with an always modality $\Box$, and in **GPS-WRITE** by requiring the user to establish the interpretation of the new write using only the local resource P . In contrast, iGPS does not provide the user with the full interpretation, but only the read interpretation τ_{read} . This weakens, for example, the **iGPS-WRITE** rule in comparison with **GPS-WRITE**, because the user cannot use τ_{full} to show $s' \sqsubseteq s''$.

The reason for this discrepancy between GPS and iGPS is that the soundness proof of GPS reasons about an entire program execution graph at once. With its bird’s-eye view of the entire execution, GPS can, for the duration of a step, assemble resources that have been transferred elsewhere in the graph to re-construct τ_{full} for the user. The soundness of iGPS, on the other hand, is established in a simpler and more local manner, without involving reasoning about the full execution of the program. We avoid the global soundness argument of GPS and instead provide a pragmatic solution which—judging from our success in porting GPS examples—is effectively as strong as GPS and, at the same time, makes for a very simple soundness proof: we maintain the duplicable τ_{read} for all (past) events and can thus easily provide it to the client of **iGPS-READ**.

Essentially, the reason our rules are as effective (if not more so) than those of GPS is that GPS provides one-size-fits-all rules, which are applicable to both programs with write-write races and those without, whereas we provide special support for the common case where there are no write-write races. For programs without those races, the rules provided by GPS are quite cumbersome to use and often require additional ghost state for bookkeeping. iGPS instead supports an optimized write rule for the common case in which there are no such races, via single-writer protocols. For the remaining cases, the rather simple-minded rule **iGPS-WRITE** appears to suffice in all the examples we have considered thus far.

Our operational semantics for RA draws heavily on Lahav *et al.*’s semantics for SRA [LGV16]—a stronger variant of RA, which is equivalent to it in the absence of write-write races. SRA was developed to provide an intuitive operational characterization which is as efficiently implementable as RA. However, as we observe here, moving to an operational charac-

terization does not in fact require any strengthening of the RA semantics (even the slight strengthening of SRA). The main difference between the operational semantics of SRA and the one we give for RA is that writes in SRA always take a globally maximal timestamp, whereas in RA they need not do so. The canonical example demonstrating this difference is the 2+2W example (see Lahav *et al.* [LGV16] for more details).

Going beyond Lahav *et al.*, we offer the first operational account of the interaction of RA and non-atomic accesses. Our semantics corresponds to C11’s for programs that do not mix atomic RMW operations and non-atomic reads at the same location. We feel this is a reasonable restriction, given that C11’s treatment of programs mixing atomic and non-atomic accesses is already known to be problematic [Bat+15]. Our semantics does not correspond to C11’s for arbitrary programs, as evidenced by the following example:

$$\text{cas}(x, 0, 1) \parallel \begin{cases} x_{[\text{at}]} := 2; \\ a := x_{[\text{na}]} \end{cases}$$

C11 considers this program racy because, if the CAS succeeds, the first thread’s update of x to 1 and the second thread’s non-atomic read of x are not in a happens-before relation. In contrast, our semantics does not consider this a race because the non-atomic read is always guaranteed to read from the previous write with value 2. We find the C11 semantics for this program to be rather unintuitive, but leave a more thorough investigation of the issue to future work.

Our RA semantics may also be considered a close derivative of Kang *et al.*’s “promising” semantics [Kan+17], which is geared toward solving a broader problem with the full C11 model (the so-called “out-of-thin-air” problem). We look forward to using Iris to construct program logics for this promising semantics.

[Bat+15] Batty et al., “The Problem of Programming Language Concurrency Semantics”

[Kan+17] Kang et al., “A promising semantics for relaxed-memory concurrency”

ACKNOWLEDGEMENTS

We would like to thank Mohit Vyas for spotting a mistake in our original proof of correspondence to C11, and Mark Batty for helpful conversations.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, funded under the European Union’s Horizon 2020 Framework Programme (grant agreement no. 683289).

Appendices

A

Operational Semantics for RA+NA

We assume that programs do not include atomic update (RMW) to a location that is also read non-atomically.

A.1 DECLARATIVE SEMANTICS

We define threadpool reduction that generates *traces*:

$$\begin{array}{c}
 \text{Trace-Red-Silent} \\
 \frac{\mathcal{TS}(\pi) \rightarrow e, \text{nil}}{\mathcal{TS} \xRightarrow{\epsilon}^{\pi} \mathcal{TS}[\pi \mapsto e]} \\
 \\
 \text{Trace-Red-Mem} \\
 \frac{\mathcal{TS}(\pi) \xrightarrow{\epsilon} e, \text{nil}}{\mathcal{TS} \xRightarrow{\epsilon}^{\pi} \mathcal{TS}[\pi \mapsto e]} \\
 \\
 \text{Trace-Red-Fork} \\
 \frac{\mathcal{TS}(\pi) \xrightarrow{\langle \text{Fork}, \rho \rangle} e, e_f}{\mathcal{TS} \xRightarrow{\langle \text{Fork}, \rho \rangle}^{\pi} \mathcal{TS}[\pi \mapsto e] \uplus [\rho \mapsto e_f]}
 \end{array}$$

We write $\mathcal{TS} \Rightarrow^{\pi} \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some transition label x ; $\mathcal{TS} \xRightarrow{x} \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some thread identifier π ; and $\mathcal{TS} \Rightarrow \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some transition label x and thread identifier π . A threadpool is called *final* if $\mathcal{TS}(\pi) \in \text{Val}$ for every $\pi \in \text{dom}(\mathcal{TS})$.

Definition 1. A *trace* is a sequence of pairs $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$. We say that $tr = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ is a trace of an expression e if

$$[0 \mapsto e] \xRightarrow{\epsilon}^* \xRightarrow{\varepsilon_1}^{\pi_1} \xRightarrow{\epsilon}^* \dots \xRightarrow{\epsilon}^* \xRightarrow{\varepsilon_n}^{\pi_n} \xRightarrow{\epsilon}^* \mathcal{TS}$$

for some thread π and threadpool $\mathcal{TS}$. When $\mathcal{TS}$ is final, we call tr a *full trace*.

Lemma A.1.1 (Receptiveness). *Let $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ be a trace of an expression e .*

- If $\varepsilon_n = \langle \text{Read}_{\alpha}, \ell, v \rangle$, then for every value v' , $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \langle \text{Read}_{\alpha}, \ell, v' \rangle, \pi_n \rangle$ is a trace of e .
- If $\varepsilon_n = \langle \text{Update}, \ell, v_o, v_n \rangle$, then for every value v'_o , either $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \langle \text{Read}_{\text{at}}, \ell, v'_o \rangle, \pi_n \rangle$ is a trace of e ,

or there exists a value v'_n such that $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \langle \text{Update}, \ell, v'_o, v'_n \rangle, \pi_n \rangle$ is a trace of e .

Definition 2. A trace $tr = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ induces a partial order $\text{sb}(tr)$ (sequenced-before) on $\{1, \dots, n\}$ that is given by:

$$\frac{k_1 < k_2 \quad \pi_{k_1} = \pi_{k_2}}{\langle k_1, k_2 \rangle \in \text{sb}(tr)} \quad \frac{k_1 < k_2 \quad \varepsilon_{k_1} = \langle \text{Fork}, \pi_{k_2} \rangle}{\langle k_1, k_2 \rangle \in \text{sb}(tr)}$$

$$\frac{\langle k_1, k_2 \rangle \in \text{sb}(tr) \quad \langle k_2, k_3 \rangle \in \text{sb}(tr)}{\langle k_1, k_3 \rangle \in \text{sb}(tr)}$$

Lemma A.1.2. Let tr be a trace of an expression e . Then:

- Any prefix of tr is a trace of e .
- Any permutation tr' of tr that respects $\text{sb}(tr)$ (i.e., $\text{sb}(tr') = \text{sb}(tr)$) is a trace of e .

Definition 3. An execution graph G consists of:

1. a finite set of nodes $E \subseteq \mathbb{N}$ (called *events*).
2. a function tid assigning a thread identifier to every event in E . We denote by E^π the set of events in $a \in E$ with $\text{tid}(a) = \pi$.
3. a function lab assigning a *label* to every event in E . Labels take one of the following forms: $\langle \text{Read}_\alpha, \ell, v \rangle$, $\langle \text{Write}_\alpha, \ell, v \rangle$, $\langle \text{Update}, \ell, v_o, v_n \rangle$, or $\langle \text{Fork}, \pi \rangle$, where $\alpha \in \{\text{at}, \text{na}\}$ is the access mode, ℓ is the location accessed, and v, v_o, v_n are read/written values, and π is a thread identifier. The function lab naturally induces functions typ , mod , loc , val_r , and val_w that return (when applicable) the type (FORK, R, W or U), access mode (in particular, $\text{mod}(a) = \text{at}$ if $\text{typ}(a) = \text{U}$), location, and read/written values of an event. For $T \in \{\text{S}, \text{R}, \text{W}, \text{U}\}$, T denotes the set $\{a \in E \mid \text{typ}(a) = T\}$. We also concatenate the event sets notations, use subscripts to denote the accessed location, and superscripts for access modes (e.g., $\text{WU}_\ell^{\text{at}}$ denotes all events $a \in \text{W} \cup \text{U}$ with $\text{loc}(a) = \ell$ and $\text{mod}(a) = \text{at}$).
4. a strict partial order $\text{sb} \subseteq E \times E$, called *sequenced-before*, (a.k.a. *program order*).
5. a binary relation $\text{rf} \subseteq E \times E$ called *reads-from*, satisfying: (i) $a \in \text{WU}$ and $b \in \text{RU}$ for every $\langle a, b \rangle \in \text{rf}$; (ii) $\text{loc}(a) = \text{loc}(b)$ and $\text{val}_w(a) = \text{val}_r(b)$ for every $\langle a, b \rangle \in \text{rf}$; and (iii) $a_1 = a_2$ whenever $\langle a_1, b \rangle, \langle a_2, b \rangle \in \text{rf}$.
6. a strict partial order mo on WU , called *modification order*, which is a disjoint union of relations $\{\text{mo}_\ell\}_{\ell \in \text{Loc}}$, such that each mo_ℓ is a strict total order on WU_ℓ .

In what follows, to resolve ambiguities, we may include a prefix “ G .” to refer to the components of an execution graph G .

Definition 4. An execution graph G follows a trace $tr = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ if the following hold:

- E consists of n events $a_1, \dots, a_n$ with $\text{lab}(a_k) = \varepsilon_k$ and $\text{tid}(a_k) = \pi_k$ for every $1 \leq k \leq n$.
- $\text{sb} = \{\langle a_{k_1}, a_{k_2} \rangle \mid \langle k_1, k_2 \rangle \in \text{sb}(tr)\}$.

We call G is an *execution graph of an expression e* if G follows some trace of e .

Note that **rf** and **mo** are not restricted.

Definition 5. Given an execution graph G , the relations **sw** (*synchronization*) and **hb** (*happens-before*) are given by:

$$\text{sw} = \{\langle a, b \rangle \in \text{rf} \mid \text{mod}(a) = \text{mod}(b) = \text{at}\} \quad \text{hb} = (\text{sb} \cup \text{sw})^+$$

Definition 6 (Consistency). An execution graph G is *consistent* if the following hold:

- For every $b \in \text{RU}$, if there exists some $a \in \text{WU}_{\text{loc}(b)}$ such that $\langle a, b \rangle \in \text{hb}^?$; **sb**, then $\langle c, b \rangle \in \text{rf}$ for some $c \in E$.
- **mo**; **rf**; **hb** is irreflexive
- **mo**; **hb** is irreflexive.
- **mo**; **hb**; **rf**⁻¹ is irreflexive.
- **sb** $\cup$ **rf** is acyclic.
- **mo**; **rf**; **hb**; **rf**⁻¹ is irreflexive.
- **rf**; **hb** is irreflexive.
- $(\text{rf}; [\text{U}]) \cap (\text{mo}; \text{mo}) = \emptyset$.

Definition 7. A trace tr is *allowed* for an expression e if tr is a trace of e and there exists a consistent execution graph G that follows tr .

Definition 8 (Conflict). Two events a and b are called *conflicting* in an execution graph G if $a, b \in E$, $\{\text{typ}(a), \text{typ}(b)\} \cap \{\text{W}, \text{U}\} \neq \emptyset$, $a \neq b$, and $\text{loc}(a) = \text{loc}(b)$.

Definition 9 (Races). A pair $\langle a, b \rangle$ is called a *race* in G if a and b are conflicting events in G , and $\langle a, b \rangle \notin \text{hb} \cup \text{hb}^{-1}$. An execution graph G is called *racy* if there is some race $\langle a, b \rangle$ in G with $\text{na} \in \{\text{mod}(a), \text{mod}(b)\}$.

Definition 10 (Initialization). A location ℓ is called *uninitialized* in an execution graph G if there exists some $b \in \text{RU}_\ell$ such that $\langle a, b \rangle \notin \text{hb}^?$; **sb** for every $a \in \text{WU}_\ell$. An execution graph G is called *uninitialized* if some location is uninitialized in G .

Definition 11 (Buggy expression). An expression e is called *buggy* if some consistent execution graph of e is either racy or uninitialized.

The following definition and lemma are useful in the sequel.

Definition 12. Given an execution graph G , we denote by mo^{na} the relation $\text{mo} \cap ((\text{W}^{\text{na}} \times E) \cup (E \times \text{W}^{\text{na}}))$.

Lemma A.1.3. Let G be a non-racy consistent execution graph. Then, $\text{hb} = (\text{sb} \cup \text{rf} \cup \text{mo}^{\text{na}})^+$.

Proof. Since G is not racy, we have $\text{rf} \cup \text{mo}^{\text{na}} \subseteq \text{hb} \cup \text{hb}^{-1}$. Since G is consistent, **rf**; **hb** and mo^{na} ; **hb** are irreflexive, and so $\text{rf} \cup \text{mo}^{\text{na}} \subseteq \text{hb}$. It easily follows that $\text{hb} = (\text{sb} \cup \text{rf} \cup \text{mo}^{\text{na}})^+$. $\square$

A.1.1 Alternative declarative semantics

Next, we present an alternative declarative semantics in which **hb** is substituted by $(\text{sb} \cup \text{rf})^+$, and races require stronger conditions as defined below. This semantics simplifies our proof below of the equivalence to the operational semantics.

Definition 13 (Strong consistency). *Strong consistency* is defined exactly as consistency with $(\text{sb} \cup \text{rf})^+$ instead of **hb**.

Observing that $\text{hb} \subseteq (\text{sb} \cup \text{rf})^+$ in every execution graph, it is easy to see that strong consistency implies consistency. In addition, strong consistency can be simplified as follows:

Lemma A.1.4. *An execution graph G is strongly consistent iff the following hold:*

- For every $b \in \text{RU}$, if there exists some $a \in \text{WU}_{\text{loc}(b)}$ such that $\langle a, b \rangle \in (\text{sb} \cup \text{rf})^*; \text{sb}$, then $\langle c, b \rangle \in \text{rf}$ for some $c \in \text{E}$.
- $\text{mo}; (\text{sb} \cup \text{rf})^+$ is irreflexive.
- $\text{mo}; (\text{sb} \cup \text{rf})^+; \text{rf}^{-1}$ is irreflexive.
- $\text{sb} \cup \text{rf}$ is acyclic.
- $(\text{rf}; [\text{U}]) \cap (\text{mo}; \text{mo}) = \emptyset$.

Definition 14 (Strongly racy). An execution graph G is called *strongly racy* if there exist $a, b \in \text{E}$ such that the following hold:

- $a \neq b$.
- $\text{loc}(a) = \text{loc}(b)$.
- $\langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$ and $\langle b, a \rangle \notin (\text{sb} \cup \text{rf})^+$.
- Either $(a \in \text{W}^{\text{at}}$ and $b \in \text{R}^{\text{na}})$ or $a \in \text{W}^{\text{na}}$.

Definition 15 (Strong initialization). A location ℓ is called *strongly uninitialized* in an execution graph G if there exists some $b \in \text{RU}_\ell$ such that $\langle a, b \rangle \notin (\text{sb} \cup \text{rf})^*; \text{sb}$ for every $a \in \text{WU}_\ell$. An execution graph G is called *strongly uninitialized* if some location is strongly uninitialized in G .

Notation 1. Let G be an execution graph. For a set $E \subseteq \text{E}$, we denote by $G \cap E$ the execution graph G' given by: $G'.\text{E} = E$, $G'.\text{tid} = G.\text{tid}|_E$, $G'.\text{lab} = G.\text{lab}|_E$, $G'.\text{sb} = G.\text{sb} \cap (E \times E)$, $G'.\text{rf} = G.\text{rf} \cap (E \times E)$, and $G'.\text{mo} = G.\text{mo} \cap (E \times E)$. Given $a \in \text{E}$, we write $G \setminus \{a\}$ for $G \cap (E \setminus \{a\})$. Given $R \subseteq E \times E$, by R -prefix of G we mean any execution graph G' such that $G' = G \cap E$ for some set E satisfying $a \in E$ whenever $b \in E$ and $\langle a, b \rangle \in R$.

Lemma A.1.5 (Adding writes). Let G be an execution graph and $a \in \text{W}$ be an $(\text{sb} \cup \text{rf})^+$ -maximal event in E . Suppose that $G \setminus \{a\}$ is strongly consistent. Let G' be the execution graph that is obtained from G by making a an **mo**-maximal event. Then, G' is strongly consistent.

Lemma A.1.6 (Adding reads). *Let G be an execution graph and $a \in \mathbb{R}$ be an $(\text{sb} \cup \text{rf})^+$ -maximal event in E . Suppose that $G \setminus \{a\}$ is strongly consistent, and that $\langle b, a \rangle \in (\text{sb} \cup \text{rf})^*; \text{sb}$ for some $b \in \mathbb{WU}_{\text{loc}(a)}$. Then, there exists a strongly consistent execution graph G' that satisfies the following:*

- $G'.E = G.E$, $G'.\text{tid} = G.\text{tid}$, $G'.\text{sb} = G.\text{sb}$, and $G'.\text{mo} = G.\text{mo}$.
- $G'.\text{lab}(b) = G.\text{lab}(b)$ for every $b \in G.E \setminus \{a\}$
and $G'.\text{lab}(a) = \langle \text{Read}_{G.\text{mod}(a)}, G.\text{loc}(a), v \rangle$ for some value v .
- $G'.\text{rf} \cap (G.E \times (G.E \setminus \{a\})) = G.\text{rf} \cap (G.E \times (G.E \setminus \{a\}))$.
- $G'.\text{rf} \cap (G.E \times \{a\}) \subseteq (G'.\text{sb} \cup G'.\text{rf})^*; G'.\text{sb}$.

Proof. Let $B = \{b \in \mathbb{WU}_{\text{loc}(a)} \mid \langle b, a \rangle \in (\text{sb} \cup \text{rf})^*; \text{sb}\}$, and let $b = \max_{\text{mo}} B$. Let G' be the execution graph obtained from G by replacing the value read at a by $\text{val}_w(b)$, and adding the rf -edge $\langle b, a \rangle$ (instead of the rf -edge entering a in G , if such exists). It is straightforward to see that G' is a strongly consistent. $\square$

Lemma A.1.7 (Adding updates). *Let G be an execution graph and $a \in \mathbb{U}$ be an $(\text{sb} \cup \text{rf})^*$ -maximal event in E . Suppose that $G \setminus \{a\}$ is strongly consistent, and that $\langle b, a \rangle \in (\text{sb} \cup \text{rf})^*; \text{sb}$ for some $b \in \mathbb{WU}_{\text{loc}(a)}$. Then, there exists a strongly consistent execution graph G' that satisfies the following:*

- $G'.E = G.E$, $G'.\text{tid} = G.\text{tid}$, $G'.\text{sb} = G.\text{sb}$.
- $G'.\text{lab}(b) = G.\text{lab}(b)$ for every $b \in G.E \setminus \{a\}$
and $G'.\text{lab}(a) = \langle \text{Update}, G.\text{loc}(a), v, v' \rangle$ for some values v, v' .
- $G'.\text{rf} \cap (G.E \times (G.E \setminus \{a\})) = G.\text{rf} \cap (G.E \times (G.E \setminus \{a\}))$.
- $G'.\text{rf} \cap (G.E \times \{a\}) \subseteq ((G'.\text{sb} \cup G'.\text{rf})^+; G'.\text{sb}) \cup (G.\mathbb{WU}^{\text{at}} \times \{a\})$.
- $G'.\text{mo} \cap ((G.E \setminus \{a\}) \times (G.E \setminus \{a\})) = G.\text{mo} \cap ((G.E \setminus \{a\}) \times (G.E \setminus \{a\}))$.

Proof. Let $B = (\mathbb{WU}_{\ell}^{\text{at}} \setminus \{a\}) \cup \{b \in \mathbb{W}_{\ell}^{\text{na}} \mid \langle b, a \rangle \in (\text{sb} \cup \text{rf})^*; \text{sb}\}$, where $\ell = \text{loc}(a)$. Let $b = \max_{\text{mo}} B$, and let G' be the execution graph obtained from G by (i) replacing the value read at a by $\text{val}_w(b)$; (ii) adding the rf -edge $\langle b, a \rangle$ (instead of the rf -edge entering a in G , if such exists); and (iii) making a to be the mo -successor of b . It is straightforward to see that G' is a strongly consistent. $\square$

Lemma A.1.8 (Modifying updates). *Let G be a strongly consistent execution graph, and let $a \in \mathbb{U}$ be an $(\text{sb} \cup \text{rf})^+$ -maximal event in E . Then, any execution graph G' that is obtained from G by changing the value written at a , or by making a a read event, is strongly consistent.*

Lemma A.1.9. *Let G be a racy consistent execution graph of an expression e . Then, some strongly consistent execution graph of e is either strongly racy or strongly uninitialized.*

Proof. Let $a_1, \dots, a_n$ be an enumeration of E that respects $\text{sb} \cup \text{rf}$, and let k be the minimal index such that $G \cap \{a_1, \dots, a_k\}$ is racy.

First, we claim that for every event a , if $\langle b, a_k \rangle \in (\text{sb} \cup \text{rf} \cup \text{mo}^{\text{na}})^*; \text{sb}$, then $\langle b, a_k \rangle \in \text{hb}$. Indeed, suppose that $\langle b, a_k \rangle \in (\text{sb} \cup \text{rf} \cup \text{mo}^{\text{na}})^*; \text{sb}$. Let $G' = G \cap \{a_1, \dots, a_{k-1}\}$. The minimality of k ensures that G' is not racy, and by Lemma A.1.3, it follows that $\langle b, a_k \rangle \in G.\text{hb}^?; G.\text{sb} \subseteq G.\text{hb}$.

Next, let j be the minimal index such that $\langle a_j, a_k \rangle$ is a race in G and $\text{na} \in \{\text{mod}(a_j), \text{mod}(a_k)\}$. Let $E = \{a \in E \mid \langle a, a_k \rangle \in (\text{sb} \cup \text{rf})^* \vee \langle a, a_j \rangle \in (\text{sb} \cup \text{rf})^*\}$, and let $G_0 = G \cap E$. Note that a_j has no outgoing $\text{sb} \cup \text{rf}$ -edges in G_0 except (possibly) an rf -edge to a_k (otherwise, $\langle a_j, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^*; G.\text{sb}$, and so $\langle a_j, a_k \rangle \in G.\text{hb}$, which contradicts the fact that $\langle a_j, a_k \rangle$ is a race in G). It is also easy to verify that G_0 is a strongly consistent execution graph of e . If G_0 is strongly uninitialized, then we are done. Otherwise, if $a_k \in G_0.\text{RU}$ then, we have $\langle b, a_k \rangle \in (G_0.\text{sb} \cup G_0.\text{rf})^*; G_0.\text{sb}$ for some $b \in G_0.\text{WU}_{\text{loc}(a_k)}$. Consider the possible cases:

- $a_k \in \text{R}$: Then, we must have $a_j \in \text{W}$. Let G' be a strongly consistent execution graph that satisfies the properties mentioned in Lemma A.1.6 applied on G_0 and a_k . The receptiveness property ensures that G' is an execution graph of e . Let G'' be a strongly consistent execution graph that satisfies the properties mentioned in Lemma A.1.5 applied on G' and a_j . Since $G''.E = G'.E$, $G''.\text{lab} = G'.\text{lab}$ and $G''.\text{sb} = G'.\text{sb}$, we have that G'' is an execution graph of e as well. It follows that $\langle a_j, a_k \rangle \notin G''.\text{mo}^?; (G''.\text{sb} \cup G''.\text{rf})^+$, and $\langle a_k, a_j \rangle \notin (G''.\text{sb} \cup G''.\text{rf})^+$. In addition, since $\langle a_j, a_k \rangle$ is a race in G , we have $G''.\text{loc}(a_j) = G''.\text{loc}(a_k)$. Hence, G'' is a strongly racy strongly consistent execution graph of e .
- $a_k \in \text{W}$: Let G' be a strongly consistent execution graph that satisfies the properties mentioned in Lemma A.1.5 applied on G_0 and a_k . As in the previous case, G' is an execution graph of e . It follows that $\langle a_k, a_j \rangle \notin G'.\text{mo}^?; (G'.\text{sb} \cup G'.\text{rf})^+$, and $\langle a_j, a_k \rangle \notin (G'.\text{sb} \cup G'.\text{rf})^+$. In addition, since $\langle a_j, a_k \rangle$ is a race in G , we have $G'.\text{loc}(a_j) = G'.\text{loc}(a_k)$. Hence, G' is a strongly racy strongly consistent execution graph of e .
- $a_k \in \text{U}$: Then, we must have $a_j \in \text{W}^{\text{na}}$. Let G' be a strongly consistent execution graph that satisfies the properties mentioned in Lemma A.1.7 applied on G_0 and a_k . The receptiveness property ensures that some execution graph G^* obtained from G' by possibly making a_k a read event is an execution graph of e . By Lemma A.1.8, G^* is also strongly consistent. Note also that we do not have $\langle a_j, a_k \rangle \in G^*.\text{rf}$. Let G'' be a strongly consistent execution graph that satisfies the properties mentioned in Lemma A.1.5 applied on G^* and a_j . Since $G''.E = G^*.E$, $G''.\text{lab} = G^*.\text{lab}$ and $G''.\text{sb} = G^*.\text{sb}$, we have that G'' is an execution graph of e as well. It follows that $\langle a_j, a_k \rangle \notin G''.\text{mo}^?; (G''.\text{sb} \cup G''.\text{rf})^+$, and $\langle a_k, a_j \rangle \notin (G''.\text{sb} \cup G''.\text{rf})^+$. In addition, since $\langle a_j, a_k \rangle$ is a race in G , we have $G''.\text{loc}(a_j) = G''.\text{loc}(a_k)$. Hence, G'' is a strongly racy strongly consistent execution graph of e . $\square$

Lemma A.1.10. *Let G be an uninitialized consistent execution graph of an expression e . Then, some strongly consistent execution graph of e is either strongly racy or strongly uninitialized.*

Proof. Let ℓ be an uninitialized location in G , and let $b \in \text{RU}_\ell$ such that $\langle a, b \rangle \notin \text{hb}^?; \text{sb}$ for every $a \in \text{WU}_\ell$. If G is racy, then we are done by Lemma A.1.9. Otherwise, by Lemma A.1.3, we have $\text{hb} = (\text{sb} \cup \text{rf} \cup \text{mo}^{\text{na}})^+$. Hence, $\langle a, b \rangle \notin (\text{sb} \cup \text{rf})^*; \text{sb}$ for every $a \in \text{WU}_\ell$, and so G is strongly uninitialized. $\square$

Corollary I.2. *Let e be an expression. Then:*

- *If e is not buggy, then a trace tr is allowed for e iff tr is a trace of e and there exists a strongly consistent execution graph G that follows tr .*
- *e is buggy iff some strongly consistent execution graph of e is either strongly racy or strongly uninitialized.*

Proof. Easily follows from our definitions, Lemma A.1.9, Lemma A.1.10, and Lemma A.1.3:

- Suppose that e is not buggy. If tr is a trace of e , and G is a strongly consistent execution graph that follows tr , then G is also consistent, and, by definition, tr be a trace that is allowed for e . For the converse, let tr be a trace that is allowed for e . Then, by definition, tr is a trace of e and there exists a consistent execution graph G that follows tr . Since e is not buggy, G is not racy. By Lemma A.1.3, we have $(G.\text{sb} \cup G.\text{rf})^+ \subseteq G.\text{hb}$, and hence G is strongly consistent.
- If some strongly consistent execution graph G of e is either strongly racy or strongly uninitialized, then G is also a consistent execution graph of e which is either racy or uninitialized, and hence e is buggy. For the converse, suppose that e is buggy. By definition, there exists a consistent execution graph G of e that is either racy or uninitialized. Then, by Lemma A.1.9 and Lemma A.1.10, some strongly consistent execution graph of e is either strongly racy or strongly uninitialized. $\square$

Lemma A.1.11. *Let G be a strongly consistent execution graph that is not strongly racy. Then, $\text{mo}^{\text{na}} \subseteq (\text{sb} \cup \text{rf})^+$.*

Proof. Let $\langle a, b \rangle \in \text{mo}^{\text{na}}$. First, if $b \in \text{W}^{\text{na}}$, then since G is not strongly racy, and $\text{mo}; (\text{sb} \cup \text{rf})^+$ is irreflexive, we must have $\langle a, b \rangle \in (\text{sb} \cup \text{rf})^+$. Otherwise, we have $a \in \text{W}^{\text{na}}$. Let $b_1, \dots, b_n$ be an enumeration of E that respects $\text{sb} \cup \text{rf}$. We prove by induction on k that $\langle a, b_k \rangle \in \text{mo}^{\text{na}}$ implies $\langle a, b_k \rangle \in (\text{sb} \cup \text{rf})^+$. Suppose the claim hold for all $k < m$, we prove it for m . Assume that $\langle a, b_m \rangle \in \text{mo}^{\text{na}}$. Since G is not strongly racy and $\text{mo}; (\text{sb} \cup \text{rf})^+$ is irreflexive, we have $\langle a, b_m \rangle \in \text{mo}^?; (\text{sb} \cup \text{rf})^+$. Let $b_i \in \text{E}$ such that $\langle a, b_i \rangle \in \text{mo}^?$ and $\langle b_i, b_m \rangle \in (\text{sb} \cup \text{rf})^+$. If $a = b_i$, then we are done. Otherwise, by the induction hypothesis, $\langle a, b_i \rangle \in (\text{sb} \cup \text{rf})^+$, and again it follows that $\langle a, b_m \rangle \in (\text{sb} \cup \text{rf})^+$. $\square$

A.2 FIRST OPERATIONAL SEMANTICS

We present an operational semantics, equivalent to the declarative RA+NA semantics, in the form of an abstract machine whose states are execution

graphs, incrementally growing during the run. The initial execution graph, denoted G_0 , is the empty execution graph.

Notation 2. For two execution graphs G and G' , thread identifiers π, ρ , and a label ε , we write $G' \in \text{Add}(G, \pi, \rho, \varepsilon)$ if there exists an event a such that the following hold:

- $G'.\mathbf{E} = G.\mathbf{E} \uplus \{a\}$, $G'.\text{tid} = G.\text{tid} \cup \{a \mapsto \rho\}$, and $G'.\text{lab} = G.\text{lab} \cup \{a \mapsto \varepsilon\}$.
- $G'.\text{sb} = (G.\text{sb} \cup (G.\mathbf{E}^\pi \times \{a\}))^+$.
- $G'.\text{rf} \supseteq G.\text{rf}$.
- $G'.\text{mo} \supseteq G.\text{mo}$ and if $\varepsilon = \langle \text{Write}_{na}, -, - \rangle$, then a is $G'.\text{mo}$ -maximal.

The machine steps are based on execution graph steps:

$$\begin{array}{c}
 \varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Write}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \\
 \forall a \in \mathbb{W}_\ell^{\text{na}}. \exists b \in \mathbf{E}^\pi. \langle a, b \rangle \in \text{mo}^?; (\text{sb} \cup \text{rf})^* \\
 \varepsilon = \langle \text{Read}_{na}, \ell, v \rangle \Rightarrow \forall a \in \mathbb{W}_\ell. \exists b \in \mathbf{E}^\pi. \langle a, b \rangle \in \text{mo}^?; (\text{sb} \cup \text{rf})^* \\
 \varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \Rightarrow \exists a \in \mathbb{WU}_\ell, b \in \mathbf{E}^\pi. \langle a, b \rangle \in (\text{sb} \cup \text{rf})^* \\
 G' \in \text{Add}(G, \pi, \pi, \varepsilon) \quad G' \text{ is strongly consistent} \\
 \hline
 G \xrightarrow{\varepsilon}^\pi G'
 \end{array}$$

$$\begin{array}{c}
 G' \in \text{Add}(G, \pi, \rho, \langle \text{Fork}, \rho \rangle) \\
 \hline
 G \xrightarrow{\langle \text{Fork}, \rho \rangle}^\pi G'
 \end{array}$$

Race-I

$$\begin{array}{c}
 \varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Write}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \\
 \exists a \in \mathbb{W}_\ell^{\text{na}}. \forall b \in \mathbf{E}^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^* \\
 \hline
 G \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}
 \end{array}$$

Race-II

$$\begin{array}{c}
 \exists a \in \mathbb{W}_\ell. \forall b \in \mathbf{E}^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^* \\
 \hline
 G \xrightarrow{\langle \text{Read}_{na}, \ell, v \rangle}^\pi \perp_{\text{race}}
 \end{array}$$

Uninitialized

$$\begin{array}{c}
 \varepsilon \in \{ \langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle \} \\
 \neg \exists a \in \mathbb{WU}_\ell, b \in \mathbf{E}^\pi. \langle a, b \rangle \in (\text{sb} \cup \text{rf})^* \\
 \hline
 G \xrightarrow{\varepsilon}^\pi \perp_{\text{uninit}}
 \end{array}$$

The machine steps are combined with expression reductions and lifted to thread-pool reductions exactly as in [Figure 3.4](#) (using execution graphs G instead of physical states σ), defining reductions of the form $G; \mathcal{TS} \xrightarrow{x}^\pi G'; \mathcal{TS}'$. As before, we write $G; \mathcal{TS} \rightarrow^\pi G'; \mathcal{TS}'$ if $G, \mathcal{TS} \xrightarrow{x}^\pi G', \mathcal{TS}'$ for some transition label x ; $G; \mathcal{TS} \xrightarrow{\pi} G'; \mathcal{TS}'$ if $G, \mathcal{TS} \xrightarrow{x}^\pi G', \mathcal{TS}'$ for some thread identifier π ; and $G, \mathcal{TS} \rightarrow G', \mathcal{TS}'$ if $G, \mathcal{TS} \xrightarrow{x}^\pi G', \mathcal{TS}'$ for some transition label x and thread identifier π .

Lemma A.2.1. Suppose that $G_0 \xrightarrow{\varepsilon_1}^{\pi_1} \dots \xrightarrow{\varepsilon_n}^{\pi_n} G$ for some trace $tr = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$. Then G is a strongly consistent execution graph that follows tr .

Proof. Easily follows from our definitions (note that strong consistency is trivially preserved in fork steps). $\square$

Lemma A.2.2. *Let G be a strongly consistent execution graph of an expression e . Suppose that G is not strongly racy or strongly uninitialized. Then, there exists a trace $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e , such that $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G$.*

Proof. By Lemma A.1.11, $G.\text{sb} \cup G.\text{rf} \cup G.\text{mo}^{\text{na}}$ is acyclic. Let $a_1, \dots, a_n$ be an enumeration of $G'.E$ that respects $(G'.\text{sb} \cup G'.\text{rf} \cup G'.\text{mo}^{\text{na}})^+$. For every $1 \leq i \leq n$, let $\pi_i = \text{tid}(a_i)$ and $\varepsilon_i = \text{lab}(a_i)$. Then, $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ is a trace of e , and it is easy to see that by following this enumeration, i.e., adding a_i in step i for every $1 \leq i \leq n$, we obtain that $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G'$. In particular, note that the fact that G is strongly consistent entails that any $(G.\text{sb} \cup G.\text{rf})^+$ -prefix of G is also strongly consistent. $\square$

Theorem I.3. *Let e be a non-buggy expression. Then, a trace tr is allowed for e iff tr has a permutation $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ that respects $\text{sb}(tr)$, and*

$$G_0; [0 \mapsto e] \xrightarrow{\epsilon}^* \xrightarrow{\varepsilon_1} \pi_1 \xrightarrow{\epsilon}^* \dots \xrightarrow{\epsilon}^* \xrightarrow{\varepsilon_n} \pi_n \xrightarrow{\epsilon}^* G; \mathcal{TS}$$

for some execution graph G and threadpool $\mathcal{TS}$.

Proof. Follows from our definitions, Cor. I.2, Lemma A.2.1 and Lemma A.2.2:

$(\Rightarrow)$ Suppose that a trace tr is allowed for e . By Cor. I.2, tr is a trace of e and there exists a strongly consistent execution graph G that follows tr . Furthermore, Cor. I.2 entails that G is not strongly racy or strongly uninitialized. By Lemma A.2.2, there exists a trace $tr' = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e , such that $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G$. By definition, tr' must be a permutation of tr that respects $\text{sb}(tr)$.

$(\Leftarrow)$ Suppose that tr has a permutation $tr' = \langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ that respects $\text{sb}(tr)$, and $G_0; [0 \mapsto e] \xrightarrow{\epsilon}^* \xrightarrow{\varepsilon_1} \pi_1 \xrightarrow{\epsilon}^* \dots \xrightarrow{\epsilon}^* \xrightarrow{\varepsilon_n} \pi_n \xrightarrow{\epsilon}^* G; \mathcal{TS}$ for some execution graph G and threadpool $\mathcal{TS}$. By definition, tr' is a trace of e . By Lemma A.2.1, G is a strongly consistent execution graph that follows tr' . Since tr' is a permutation of tr that respects $\text{sb}(tr)$, G follows tr , and by Lemma A.1.2, tr is a trace of e . Hence, tr is allowed for e . $\square$

Theorem I.4. *An expression e is buggy iff $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{race}}$ or $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{uninit}}$.*

Proof.

$(\Rightarrow)$ Suppose that e is buggy. By Cor. I.2, some strongly consistent execution graph G of e is strongly racy or strongly uninitialized. Let $a_1, \dots, a_n$ be an enumeration of $G.E$ that respects $\text{sb} \cup \text{rf}$. Let k be the minimal index such that $G \cap \{a_1, \dots, a_k\}$ is strongly racy or strongly uninitialized. Consider the two cases:

- $G \cap \{a_1, \dots, a_k\}$ is strongly racy. Let $j < k$ be the minimal index such that $\text{loc}(a_k) = \text{loc}(a_j)$ and one of the following hold:
 1. $\langle a_k, a_j \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$ and $\langle a_j, a_k \rangle \notin (\text{sb} \cup \text{rf})^+$, and either $(a_k \in \mathbb{W}^{\text{at}}$ and $a_j \in \mathbb{R}^{\text{na}})$ or $a_k \in \mathbb{W}^{\text{na}}$.

2. $\langle a_j, a_k \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$ and $\langle a_k, a_j \rangle \notin (\text{sb} \cup \text{rf})^+$, and either $(a_j \in \mathbb{W}^{\text{at}}$ and $a_k \in \mathbb{R}^{\text{na}})$ or $a_j \in \mathbb{W}^{\text{na}}$.

Distinguish the following cases:

- $a_k \in \mathbb{W}^{\text{na}}$ and $\langle a_k, a_j \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$:

Let $B = \{a \in \mathbb{E} \mid \langle a, a_j \rangle \in (G.\text{sb} \cup G.\text{rf})^+ \vee \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^*\}$, and let $G' = G \cap B$. Then, G' is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma A.2.2, $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G'$ for some trace $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e . Since $\langle a_k, a_j \rangle \notin G.\text{mo}^?; (G.\text{sb} \cup G.\text{rf})^+$, we have that $\langle a_k, b \rangle \notin G'.\text{mo}^?; (G'.\text{sb} \cup G'.\text{rf})^+$ for every $b \in G'.\text{E}^{\text{tid}(a_j)}$. Since $a_k \in G'.\mathbb{W}_{\text{loc}(a_k)}^{\text{na}}$, we have $G' \xrightarrow{\text{lab}(a_j) \rightarrow \text{tid}(a_j)} \perp_{\text{race}}$. Furthermore, it is easy to see that $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle, \langle \text{lab}(a_j), \text{tid}(a_j) \rangle$ is also a trace of e , and so $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{race}}$.

- $a_j \in \mathbb{W}^{\text{na}}$ and $\langle a_j, a_k \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$:

This case is handled symmetrically.

- $a_k \in \mathbb{W}^{\text{at}}, a_j \in \mathbb{R}^{\text{na}}$, and $\langle a_k, a_j \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$:

Let $B = \{a \in \mathbb{E} \mid \langle a, a_j \rangle \in (G.\text{sb} \cup G.\text{rf})^+ \vee \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^*\}$, and let $G' = G \cap B$. Then, G' is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma A.2.2, $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G'$ for some trace $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e . Since $\langle a_k, a_j \rangle \notin G.\text{mo}^?; (G.\text{sb} \cup G.\text{rf})^+$, we have that $\langle a_k, b \rangle \notin G'.\text{mo}^?; (G'.\text{sb} \cup G'.\text{rf})^+$ for every $b \in G'.\text{E}^{\text{tid}(a_j)}$. Since $a_k \in G'.\mathbb{W}_{\text{loc}(a_k)}$, we have $G' \xrightarrow{\text{lab}(a_j) \rightarrow \text{tid}(a_j)} \perp_{\text{race}}$. Furthermore, it is easy to see that $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle, \langle \text{lab}(a_j), \text{tid}(a_j) \rangle$ is also a trace of e , and so $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{race}}$.

- $a_j \in \mathbb{W}^{\text{at}}, a_k \in \mathbb{R}^{\text{na}}$, and $\langle a_j, a_k \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^+$:

This case is handled symmetrically.

- $G \cap \{a_1, \dots, a_k\}$ is strongly uninitialized: Then, $a_k \in \mathbb{R}^{\text{U}}$ and $\langle a, a_k \rangle \notin (\text{sb} \cup \text{rf})^*; \text{sb}$ for every $a \in \mathbb{W}_{\text{loc}(a_k)}$. Let $B = \{a \in \mathbb{E} \mid \langle a, a_k \rangle \in (G.\text{sb} \cup G.\text{rf})^+\}$, and let $G' = G \cap B$. Then, G' is a $(\text{sb} \cup \text{rf})^+$ -prefix of G , it is strongly consistent, not strongly racy, and not strongly uninitialized, and by Lemma A.2.2, $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n G'$ for some trace $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e . Since $\langle a, a_k \rangle \notin (G.\text{sb} \cup G.\text{rf})^*; G.\text{sb}$ for every $a \in G.\mathbb{W}_{\text{loc}(a_k)}$, we have $G' \xrightarrow{\text{lab}(a_k) \rightarrow \text{tid}(a_k)} \perp_{\text{uninit}}$. Furthermore, it is easy to see that $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle, \langle \text{lab}(a_k), \text{tid}(a_k) \rangle$ is also a trace of e , and so $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{uninit}}$.

($\Leftarrow$) Suppose that $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{race}}$ or $G_0; [0 \mapsto e] \rightarrow^* \perp_{\text{uninit}}$. Then, there exists a trace $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ of e , such that $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{race}}$ or $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{uninit}}$. Let G_{n-1} such that $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_{n-1}} \pi_{n-1} G_{n-1} \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{race}}$, or $G_0 \xrightarrow{\varepsilon_1} \pi_1 \dots \xrightarrow{\varepsilon_{n-1}} \pi_{n-1} G_{n-1} \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{uninit}}$. By Lemma A.2.1, G_{n-1} is a strongly consistent execution of e . Since $G_{n-1} \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{race}}$ or $G_{n-1} \xrightarrow{\varepsilon_n} \pi_n \perp_{\text{uninit}}$, at least one of the following hold:

- $\varepsilon_n = \langle \text{Read}_{\text{na}}, \ell, v \rangle$ and there exists $a \in G_{n-1}.\text{WU}_\ell$, such that $\langle a, b \rangle \notin (G_{n-1}.\text{sb} \cup G_{n-1}.\text{rf})^+$ for every $b \in G_{n-1}.\text{E}^{\pi_n}$. Let $G' \in \text{Add}(G_{n-1}, \pi_n, \pi_n, \varepsilon_n)$, and let b denote the additional event in G' . Clearly, G' is an execution of e . Using Lemma A.1.6, as well as the receptiveness property of traces, we can obtain an execution G'' of e that is consistent, and satisfies $\langle a, b \rangle \notin G''.\text{hb}$. Since $\text{mod}(b) = \text{na}$, it follows that G'' is racy.
- $\varepsilon_n \in \{\langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Write}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle\}$, and there exists $a \in G_{n-1}.\text{W}_\ell^{\text{na}}$, such that $\langle a, b \rangle \notin (G_{n-1}.\text{sb} \cup G_{n-1}.\text{rf})^+$ for every $b \in G_{n-1}.\text{E}^{\pi_n}$. Let $G' \in \text{Add}(G_{n-1}, \pi_n, \pi_n, \varepsilon_n)$, and let b denote the additional event in G' . Clearly, G' is an execution of e . Using Lemma A.1.5, Lemma A.1.6 and Lemma A.1.7, as well as the receptiveness property of traces, and Lemma A.1.8 in case ε_n is an update, we can obtain an execution G'' of e that is consistent, and satisfies $\langle a, b \rangle \notin G''.\text{hb}$. Since $\text{mod}(a) = \text{na}$, it follows that G'' is racy.
- $\varepsilon_n \in \{\langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle\}$, and there do not exist $a \in G_{n-1}.\text{WU}_\ell$ and $b \in G_{n-1}.\text{E}^\pi$ such that $\langle a, b \rangle \in (G_{n-1}.\text{sb} \cup G_{n-1}.\text{rf})^*$. Let $G' \in \text{Add}(G_{n-1}, \pi_n, \pi_n, \varepsilon_n)$ with $G'.\text{rf} = G.\text{rf}$. Clearly, G' is an execution of e , and it is easy to see that it is consistent and uninitialized. $\square$

A.3 TIMESTAMP MACHINE

Next, we show that the timestamp machine presented in §3 is equivalent to the operational machine above.

Definition 16. A *timestamp assignment* for an execution graph G is a function $ts : \text{WU} \rightarrow \text{Time}$, that satisfies $ts(a) < ts(b)$ whenever $\langle a, b \rangle \in \text{mo}$. Given a timestamp assignment ts for G and an event $a \in \text{W} \cup \text{U}$, the view of an event a in G according to ts , denoted $\text{view}(a, G, ts)$, and the message induced by a in G according to ts , denoted $\text{msg}(a, G, ts)$, are given by:

$$\begin{aligned} \text{view}(a, G, ts) &= \lambda \ell. \max\{ts(b) \mid b \in \text{WU}_\ell, \langle b, a \rangle \in (\text{sb} \cup \text{rf})^*\} \\ \text{msg}(a, G, ts) &= (\text{loc}(a), \text{val}_w(a), ts(a), \text{view}(a, G, ts)) \end{aligned}$$

If $\{ts(b) \mid b \in \text{WU}_\ell, \langle b, a \rangle \in (\text{sb} \cup \text{rf})^*\}$ is empty, we take $\text{view}(a, G, ts)(\ell) = \perp$ (undefined).

Definition 17. Let G be an execution graph and ts be a timestamp assignment ts for G . The physical state $(M_G^{ts}, T_G^{ts}, N_G^{ts})$ is given by:

- $M_G^{ts} = \{\text{msg}(a, G, ts) \mid a \in \text{WU}\}$
- $T_G^{ts} = \lambda \pi. \lambda \ell. \max\{ts(a) \mid a \in \text{WU}_\ell, \exists b \in \text{E}^\pi. \langle a, b \rangle \in (\text{sb} \cup \text{rf})^*\}$
(If $\{ts(a) \mid a \in \text{WU}_\ell, \exists b \in \text{E}^\pi. \langle a, b \rangle \in (\text{sb} \cup \text{rf})^*\}$ is empty, we take $T_G^{ts}(\pi)(\ell) = \perp$)
- $N_G^{ts} = \lambda \ell. \max\{ts(a) \mid a \in \text{W}_\ell^{\text{na}}\}$ (If $\{ts(a) \mid a \in \text{W}_\ell^{\text{na}}\}$ is empty, we take $N_G^{ts}(\ell) = \perp$)

We say that G *relates* to a physical state (M, T, N) , denoted $G \sim (M, T, N)$ if $(M, T, N) = (M_G^{ts}, T_G^{ts}, N_G^{ts})$ for some timestamp assignment ts for G .

The following lemmas (tediously but) straightforwardly follow from the definitions.

Lemma A.3.1. *Suppose that $(M, T, N) \xrightarrow{\varepsilon}^\pi (M', T', N')$, and let G be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon}^\pi G'$ for some execution graph G' such that $G' \sim (M', T', N')$.*

Lemma A.3.2. *Suppose that $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$, and let G be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$.*

Lemma A.3.3. *Suppose that $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$, and let G be an execution graph such that $G \sim (M, T, N)$. Then, $G \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$.*

Lemma A.3.4. *Suppose that $G \xrightarrow{\varepsilon}^\pi G'$, and let ts' be a timestamp assignment for G' . Let $ts = ts'|_{G.\text{WU}}$. Then, ts is a timestamp assignment for G , and $(M_G^{ts}, T_G^{ts}, N_G^{ts}) \xrightarrow{\varepsilon}^\pi (M_{G'}^{ts'}, T_{G'}^{ts'}, N_{G'}^{ts'})$.*

Lemma A.3.5. *Suppose that $G_0 \xrightarrow{\varepsilon_1}^{\pi_1} \dots \xrightarrow{\varepsilon_n}^{\pi_n} G_n$. Then, $(M_0, T_0, N_0) \xrightarrow{\varepsilon_1}^{\pi_1} \dots \xrightarrow{\varepsilon_n}^{\pi_n} (M_n, T_n, N_n)$ for some physical state (M_n, T_n, N_n) .*

Proof. Let ts_n be some timestamp assignment for G_n , and for every $0 \leq k < n$, let $ts_k = ts_n|_{G_k.\text{WU}}$. For every $0 \leq k \leq n$, let $(M_k, T_k, N_k) = (M_{G_k}^{ts_k}, T_{G_k}^{ts_k}, N_{G_k}^{ts_k})$. (Clearly, this definition of (M_k, T_k, N_k) agrees with the definition of the initial physical state in [Definition 3.2.1](#)). By [Lemma A.3.4](#), we have $(M_{k-1}, T_{k-1}, N_{k-1}) \xrightarrow{\varepsilon_k}^{\pi_k} (M_k, T_k, N_k)$ for every $1 \leq i \leq n$. $\square$

Lemma A.3.6. *Suppose that $G \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$, and let (M, T, N) be a physical state such that $G \sim (M, T, N)$. Then, $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$.*

Proof. Let ts be a timestamp assignment such that $(M, T, N) = (M_G^{ts}, T_G^{ts}, N_G^{ts})$. Consider the two possible cases:

- $\varepsilon \in \{\langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Write}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle\}$, and $\exists a \in W_\ell^{\text{na}}. \forall b \in E^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^*$: We show that $T(\pi)(\ell) < N(\ell)$ (and it would follow that $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$). Let $a \in W_\ell^{\text{na}}$ be the mo -maximal event such that $\forall b \in E^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^*$. Then, $N(\ell) \geq ts(a)$. Hence, it suffices to show that $T(\pi)(\ell) < ts(a)$. Let $c \in \text{WU}_\ell$ and $b \in E^\pi$ such that $\langle c, b \rangle \in (\text{sb} \cup \text{rf})^*$. We show that $ts(c) < ts(a)$. Indeed, otherwise, we have $ts(a) \leq ts(c)$, and so $\langle a, c \rangle \in \text{mo}^?$. Hence, $\langle a, b \rangle \in \text{mo}^?; (\text{sb} \cup \text{rf})^*$, which contradicts our assumption.
- $\varepsilon = \langle \text{Read}_{\text{na}}, \ell, v \rangle$ and $\exists a \in W_\ell. \forall b \in E^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^*$: We show that $\exists v', t', V'. (\ell, v', t', V') \in M \wedge T(\pi)(\ell) < t'$ (and it would follow that $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{race}}$). Let $a \in W_\ell$ such that $\forall b \in E^\pi. \langle a, b \rangle \notin \text{mo}^?; (\text{sb} \cup \text{rf})^*$. $\text{msg}(a, G, ts) = (\ell, v', t', V')$. Then, $(\ell, v', t', V') \in M$, and it suffices to show that $T(\pi)(\ell) < t'$. Let $c \in \text{WU}_\ell$ and $b \in E^\pi$ such that $\langle c, b \rangle \in (\text{sb} \cup \text{rf})^*$. We show that $ts(c) < t' = ts(a)$. Indeed, otherwise, we have $t(c) \geq t(a)$ and so $\langle a, c \rangle \in \text{mo}^?$. Hence, $\langle a, b \rangle \in \text{mo}^?; (\text{sb} \cup \text{rf})^*$, which contradicts our assumption. $\square$

Lemma A.3.7. Suppose that $G \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$, and let (M, T, N) be a physical state such that $G \sim (M, T, N)$. Then, $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$.

Proof. Let ts be a timestamp assignment such that $(M, T, N) = (M_G^{ts}, T_G^{ts}, N_G^{ts})$. Since $G \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$, we have $\varepsilon \in \{\langle \text{Read}_\alpha, \ell, v \rangle, \langle \text{Update}, \ell, v_o, v_n \rangle\}$, and $\neg \exists a \in \text{WU}_\ell, b \in \mathbf{E}^\pi. \langle a, b \rangle \in (\text{sb} \cup \text{rf})^*$. By definition, we have $T_G^{ts}(\pi)(\ell) = \perp$, and so $(M, T, N) \xrightarrow{\varepsilon}^\pi \perp_{\text{unit}}$. $\square$

Finally, by combining the results above, we obtain the following:

Corollary I.5. Let e be an expression.

- If e is not buggy, then a trace tr is allowed for e iff tr has a permutation $\langle \varepsilon_1, \pi_1 \rangle, \dots, \langle \varepsilon_n, \pi_n \rangle$ that respects $\text{sb}(tr)$, and

$$(M_0, T_0, N_0); [0 \mapsto e] \xrightarrow{\varepsilon}^* \xrightarrow{\varepsilon_1 \rightarrow \pi_1} \xrightarrow{\varepsilon}^* \dots \xrightarrow{\varepsilon}^* \xrightarrow{\varepsilon_n \rightarrow \pi_n} \xrightarrow{\varepsilon}^* (M, T, N); \mathcal{TS}$$

for some thread π , physical state (M, T, N) , and threadpool $\mathcal{TS}$.

- e is buggy iff $(M_0, T_0, N_0); [0 \mapsto e] \rightarrow^* \perp_{\text{race}}$ or $(M_0, T_0, N_0); [0 \mapsto e] \rightarrow^* \perp_{\text{unit}}$ for some thread π .

B

Single-writer protocol proof rules

$$\begin{array}{c}
\boxed{\ell : s \mid \tau}_R \Rightarrow \square \boxed{\ell : s \mid \tau}_R \quad \boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_W \Rightarrow \perp \\
\\
\boxed{\ell : s_1 \mid \tau}_R * \boxed{\ell : s_2 \mid \tau}_R \Rightarrow s_1 \sqsubseteq s_2 \vee s_2 \sqsubseteq s_1 \\
\\
\boxed{\ell : s_1 \mid \tau}_W * \boxed{\ell : s_2 \mid \tau}_R \Rightarrow s_2 \sqsubseteq s_1 \\
\\
\boxed{\ell : s \mid \tau}_W \Rightarrow \boxed{\ell : s \mid \tau}_W * \boxed{\ell : s \mid \tau}_R \\
\\
\frac{\boxed{\ell : s \mid \tau}_W \Rightarrow \triangleright \tau_{\text{full}}(s, v) * \triangleright \tau_{\text{read}}(s, v) * Q}{\{\ell \hookrightarrow _ \} \ell_{[\text{na}]} := v \{Q\}} \\
\\
\left\{ \boxed{\ell : s \mid \tau}_W \right\} \ell_{[\text{at}]} \left\{ v. \boxed{\ell : s \mid \tau}_W * \tau_{\text{read}}(s, v) \right\} \\
\\
\frac{\forall s' \sqsupseteq s, v. P * \tau_{\text{read}}(s', v) \Rightarrow Q}{\left\{ \boxed{\ell : s \mid \tau}_R * P \right\} \ell_{[\text{at}]} \left\{ v. \exists s' \sqsupseteq s. \boxed{\ell : s' \mid \tau}_R * Q \right\}} \\
\\
\frac{P * \boxed{\ell : s'' \mid \tau}_W * \tau_{\text{full}}(s, _) \Rightarrow \tau_{\text{full}}(s'', v) * Q \quad s'' \sqsupseteq s}{\left\{ \boxed{\ell : s \mid \tau}_W * P \right\} \ell_{[\text{at}]} := v \left\{ \boxed{\ell : s'' \mid \tau}_R * Q \right\}}
\end{array}$$

Figure B.1: Single-writer protocol proof rules

B.1 ESCROW AND EXCHANGE PROOF RULES

In addition to escrows, we have mechanized a weaker variant of *exchanges*[TDV15]. Our variant is weaker in the sense that the resources put in the exchanges have to be justified at the same view. Unlike the general exchanges, our variant can be trivially proven sound. In order to enforce that the resources are put back at the same view as when they are taken out, we need to expose the *identity* of views in the logic. However,

[TDV15] Tassarotti et al., “Verifying Read-copy-update in a Logic for Weak Memory”

since we only need to know equalities of views, we just need to bind views to names and expose the names. We call those names *abstract views*. The assertion $\underline{P}_\beta$ asserts that P holds at some abstract view β . To use P a thread needs the extra knowledge $\text{Seen}(\beta)$ which states that it has observed the view with name β .

The assertion $[P \rightsquigarrow Q]_\beta^\mathcal{N}$ asserts the knowledge of an exchange at some abstract view β under $\mathcal{N}$. Escrows, unsurprisingly, are just a special instance of exchanges.

Figure B.2: Escrow and exchange proof rules.

Exchange-Intro-Left $\triangleright P \Rightarrow_{\mathcal{N}} \exists \beta. [P \rightsquigarrow Q]_\beta^\mathcal{N}$	Exchange-Intro-Right $\triangleright Q \Rightarrow_{\mathcal{N}} \exists \beta. [P \rightsquigarrow Q]_\beta^\mathcal{N}$
Exchange-Elim-Left $\frac{P * P \Rightarrow \perp}{[P \rightsquigarrow Q]_\beta^\mathcal{N} * \triangleright \underline{P}_\beta \Rightarrow_{\mathcal{N}} \triangleright \underline{Q}_\beta}$	Exchange-Elim-Right $\frac{Q * Q \Rightarrow \perp}{[P \rightsquigarrow Q]_\beta^\mathcal{N} * \triangleright \underline{Q}_\beta \Rightarrow_{\mathcal{N}} \triangleright \underline{P}_\beta}$
Escrow-Init $\triangleright Q \Rightarrow_{\mathcal{N}} [P \rightsquigarrow Q]^\mathcal{N}$	Escrow-Apply $\frac{P * P \Rightarrow \perp}{[P \rightsquigarrow Q]^\mathcal{N} * \triangleright P \Rightarrow_{\mathcal{N}} \triangleright Q}$

B.2 RSL PROOF RULES

RSL focuses on the message-passing style transferring of resources through release-write/acquire-read pairs. The two main assertions of RSL are $\text{Rel}(\ell, Q)$, representing the permission to write to ℓ , and $\text{Acq}(\ell, Q)$, representing the permission to read from ℓ . The resource $Q(v)$ is transferred through the write/read pair of value v . The assertion $\text{RMWAcq}(\ell, P)$ is another permission that is used to CAS on the location.

We present in [Figure B.3](#) the proof rules of iRSL, which were already checked by Rocq. The encoding extends the original assertions with a predicate D , which is duplicable, to represent the restricted interpretation of the underlying invariant governing ℓ (in a similar fashion to what we did in iGPS). For more details please refer to the Rocq development, which also contains a fractional variant.

Figure B.3: Proof rules of iRSL.

$$\begin{array}{c}
\text{RSL-WRel} \\
\{\triangleright Q(v) * \triangleright D(v) * \text{Rel}(\ell, Q, D)\} \ell_{[\text{at}]} := v \{ \text{Init}(\ell) * \text{Rel}(\ell, Q, D) \} \\
\\
\text{RSL-RAcq} \\
\{ \text{Init}(\ell) * \text{Acq}(\ell, R, D) \} \ell_{[\text{at}]} \{ v. \triangleright R(v) * \triangleright D(v) * \text{Acq}(\ell, R[v := \top], D) \} \\
\\
\text{RSL-CAS} \\
\frac{P * \triangleright Q(v_o) * \triangleright D(v_o) \Rightarrow_{-\mathcal{N}. \ell} \triangleright Q(v_n) * \triangleright D(v_n) * R(1, v_o) \quad \forall v \neq v_o. P * D(v) \Rightarrow R(0, v)}{\left\{ \text{Init}(\ell) * \text{RMWAcq}(\ell, Q, D) * P \right\} \mathbf{cas}(\ell, v_o, v_n) \left\{ \begin{array}{l} b. \exists v. R(b, v) * \text{Init}(\ell) \\ * \text{RMWAcq}(\ell, Q, D) \end{array} \right\}} \\
\\
\text{RSL-Rel-SplitJoin} \\
\text{Rel}(\ell, Q_1, D) * \text{Rel}(\ell, Q_2, D) \dashv\vdash \text{Rel}(\ell, \lambda v. Q_1(v) \vee Q_2(v), D) \\
\\
\text{RSL-Acq-SplitJoin} \\
\text{Acq}(\ell, R_1, D) * \text{Acq}(\ell, R_2, D) \Leftrightarrow \text{Acq}(\ell, \lambda v. R_1(v) * R_2(v), D) \\
\\
\text{RSL-RMW-Acq-Split} \\
\text{RMWAcq}(\ell, P, D) \Rightarrow \text{RMWAcq}(\ell, P, D) * \text{Acq}(\ell, \perp, D) \\
\\
\begin{array}{cc}
\text{RSL-Init-Persistent} & \text{RSL-RMW-Persistent} \\
\text{Init}(\ell) \vdash \Box \text{Init}(\ell) & \text{RMWAcq}(\ell, P, D) \vdash \Box \text{RMWAcq}(\ell, P, D)
\end{array} \\
\\
\text{RSL-Rel-Persistent} \\
\text{Rel}(\ell, Q, D) \vdash \Box \text{Rel}(\ell, Q, D) \\
\\
\text{RSL-Alloc-R} \\
\{\top\} \mathbf{alloc}(1) \{ \ell. \text{Rel}(\ell, P) * \text{Acq}(\ell, P) \} \\
\\
\text{RSL-Alloc-M} \\
\{\top\} \mathbf{alloc}(1) \{ \ell. \text{Rel}(\ell, P) * \text{RMWAcq}(\ell, P) \} \\
\\
\text{RSL-Init-R} \\
\{ \text{Own}(\ell) * \triangleright Q(v) * \triangleright D(v) \} \ell_{[\text{na}]} := v \{ \text{Init}(\ell) * \text{Rel}(\ell, Q, D) * \text{Acq}(\ell, Q, D) \} \\
\\
\text{RSL-Init-M} \\
\{ \text{Own}(\ell) * \triangleright Q(v) * \triangleright D(v) \} \ell_{[\text{na}]} := v \{ \text{Init}(\ell) * \text{Rel}(\ell, Q, D) * \text{RMWAcq}(\ell, Q, D) \}
\end{array}$$

B.3 BOUNDED TICKET LOCKS

Ticket locks are a fair locking mechanism found in the Linux kernel. The first formal proof of ticket locks in RA+NA was given in the appendix of the GPS paper. We follow that variant and its proof's intuition closely, although our proof, thanks to single-writer protocols, is significantly simpler.

Let $ns \triangleq 0, tc \triangleq 1$.

$\text{newLock}() \triangleq$ $\text{let } x = \text{alloc}(2) \text{ in}$ $x + ns_{[na]} := 0;$ $x + tc_{[na]} := 0;$ x	$\text{lock}(x) \triangleq$ $\text{let } y =$ $\text{FAI}(x + tc) \text{ in}$ repeat $\text{let } z =$ $x + ns_{[at]} \text{ in}$ $y == z$	$\text{unlock}(x) \triangleq$ $\text{let } z = x + ns_{[at]} \text{ in}$ $x + ns_{[at]} :=$ $(z + 1) \bmod C$
--	---	---

Figure B.4: Bounded ticket lock in λ_{RN} .

The code of the *bounded* ticket lock variant in λ_{RN} is given in [Figure B.4](#). The lock has two atomic fields, a *ticket* counter ($x + tc$) and a *now-serving* counter ($x + ns$), which are initialized with 0. A thread can acquire the lock by first asking for a ticket t from the ticket counter, which is a fetch-and-increment (**FAI**) operation that hands out the current value of the ticket counter as the new ticket, and increases the counter by 1. After obtaining the ticket, the thread waits for the now-serving counter to reach its ticket (it keeps checking $x + ns$), by which time it is guaranteed that previous tickets have been *served*, so it is safe to acquire the lock. To release the lock, the thread only needs to write $t + 1$ to the now-serving counter, signifying that the next ticket can now be served.

In the bounded variant of ticket locks, the counters are stored in a fixed-size machine word with the maximum unsigned value C . Once the counters reach $C - 1$, the next increment will reset them back to 0. This additional flavor of reality induces the main complication of the verification. In unbounded ticket locks, it is easy to maintain the invariant that no two threads have the same ticket. Here, however, tickets are being *reused*, and the ticket counter must ensure that a ticket t has already been served before re-issuing it. In order to circumvent this, the number of concurrent threads trying to acquire the lock is restricted at most C . Additionally, before getting a new ticket, each thread must prove to the ticket counter that its most recent ticket has been served. Consequently, the ticket counter must track which tickets have been issued to which threads and by its best the value of the now-serving counter, in order to keep the number of tickets waiting under C .

The now-serving counter is where the actual transfer of resources takes place: the lock resource P is transferred from the release write to $x + ns$ (by the **unlock** call of the previous ticket holder) to the acquire read by the **lock** call of the current ticket holder. Since a read cannot get ownership, escrows are used instead to transfer P . The releasing thread creates an escrow to store P and deliver it to the holder of ticket $t + 1$. The escrow is acquired by the $t + 1$ ticket holder when the loop of its call to **lock** stops. By that time the acquiring thread should also have the

correct exclusive token to use the escrow and obtain P . The releasing thread saves the proof that it has released the lock, in order to present that to the ticket counter when it requests another ticket later.

Our proof significantly simplifies the original proof by:

- **Using single-writer protocols for the now-serving counter.** The original proof uses a normal protocol for the counter, in combination with two other ghost components to ensure that the counter's value cannot surpass the ticket t held by the acquiring thread, and that once the lock is acquired, the value stays exactly t until the thread releases the lock by increasing it to $t + 1$. In our proof, we add the single-writer permission of $x + ns$ to the lock's resources, so a thread also acquires the full-write ownership of $x + ns$ when it acquires the lock, and releases the ownership by the same time with releasing the lock. This removes two components of the ghost monoid needed by the original proof to maintain knowledge about some upper-bound of the protocol's state.
- **Tracking only the most recent ticket of a thread.** The original proof tracks the complete history of issued tickets, which we found unnecessary. Instead, we only need to track the most recent ticket issued to a thread, which simplifies the tracking component of our ghost monoid. However, we do not claim that this results from the use of single-writer protocols: it seems that the simplification can also be applied to the original proof.

B.4 PROOF SETUP FOR BOUNDED TICKET LOCKS

Figure B.5: Proof setup for bounded ticket locks.

$$\begin{aligned}
\text{Perm}^\gamma(t) &\triangleq \left[\{ \{t\}, \circ \emptyset \} : \text{DISJ}(\mathbb{N}) \times \text{AUTH}(\mathbb{N} \xrightarrow{\text{fin}} \text{Ex}(\mathbb{N} \cup \{-1\})) \right]^\gamma \\
\text{AllTks}^\gamma(M) &\triangleq \left[\langle \emptyset, \bullet M \rangle \right]^\gamma \\
\text{MyTkt}^\gamma[i \mapsto t] &\triangleq \left[\langle \emptyset, \circ [i := t] \rangle \right]^\gamma \\
\text{Escs}(x, \gamma, n) &\triangleq \forall t \leq n. \left[\text{Perm}^\gamma(t) \rightsquigarrow P * \boxed{x + ns : t \mid \text{NSP}(\gamma)}_W \right] \\
\text{NSP}(\gamma)(n, z) &\triangleq z = n \bmod \mathbf{C} * \Box \text{Escs}(x, \gamma, n) \\
\text{Waiting}(M, n) &\triangleq \{ i \in \text{dom}(M) \mid M(i) \geq n \} \\
\text{TCP}(\gamma, x)(\text{Inv}, y) &\triangleq \exists t, n, M. y = t \bmod \mathbf{C} \wedge n \leq t \wedge t \leq n + |\text{Waiting}(M, n)| \\
&\quad \wedge t = \max_i M(i) \wedge \text{dom}(M) = [0, \dots, \mathbf{C} - 1] \\
&\quad \wedge (\forall i, j \in \text{dom}(M). M(i) = M(j) \neq -1 \Rightarrow i = j) \\
&\quad * \text{AllTks}^\gamma(M) * \bigstar_{j \geq t} \text{Perm}^\gamma(j) * \boxed{x + ns : n \mid \text{NSP}(\gamma)}_R \\
\text{MayAcquire}(x) &\triangleq \exists \gamma, i, t. \text{MyTkt}^\gamma[i \mapsto t] * \boxed{x + tc : \text{Inv} \mid \text{TCP}(\gamma, x)} \\
&\quad * t \neq -1 \Rightarrow \boxed{x + ns : t + 1 \mid \text{NSP}(\gamma)}_R \\
\text{MayRelease}(x) &\triangleq \exists \gamma, i, t. \text{MyTkt}^\gamma[i \mapsto t] * \boxed{x + tc : \text{Inv} \mid \text{TCP}(\gamma, x)} \\
&\quad * \boxed{x + ns : t \mid \text{NSP}(\gamma)}_W
\end{aligned}$$

We explain the technical construction of the proof. The complete definitions of monoids, protocols and escrows needed are given in [Figure B.5](#).

We use protocols $\mathbf{TCP}(\gamma, x)$ and $\mathbf{NSP}(\gamma)$ to govern the ticket counter and the now-serving counter respectively. The $\mathbf{TCP}(\gamma, x)$ protocol only has a single state Inv , while the states of $\mathbf{NSP}(\gamma)$ are natural numbers, which represent *abstract* tickets. Though working differently, both counters record the abstract ticket t they are currently working with, who requires the concrete counter value t_c to follow the equation $t_c = t \bmod \mathbf{C}$. The abstract ticket of each counter can only increase over time (the state relation is $\leq$ for $\mathbf{NSP}(\gamma)$). By exploiting abstract tickets, we do not need to deal with reusing concrete tickets.

The interpretations of $\mathbf{NSP}(\gamma)$ and $\mathbf{TCP}(\gamma, x)$ depend respectively on the first and second components of a product monoid for our ghost states. The first component $\text{DISJ}(\mathbb{N})$ is a *disjoint-union* monoid of natural numbers, whose composition is only defined for disjoint sets of natural numbers: $X \cdot Y = X \cup Y$ if $X \cap Y = \emptyset$. It is used for abstract ticket ownerships. The second component $\text{AUTH}(\mathbb{N} \xrightarrow{\text{fin}} \text{EX}(\mathbb{N} \cup \{-1\}))$ is an authoritative monoid of a finite partial map from $\mathbb{N}$ to exclusive elements from $\mathbb{N} \cup \{-1\}$. It is used for tracking issued abstract tickets for all participating threads. The composition works in the same way as that of the heap monoid in λ_{SC} and the history and thread-view monoids in λ_{RN} : it has the same rules for exclusiveness, agreement, and updates as those monoids. The composition of the product monoid is then lifted piecewise.

The now-serving counter protocol

The assertion $\text{Perm}^\gamma(t)$ embeds the ownership of the abstract ticket t in its first component (while the second component is an empty map which contains zero knowledge or ownership). $\text{Perm}^\gamma(t)$ satisfies exclusiveness, realizing the requirement that a ticket can only be uniquely owned:

$$\text{Perm}^\gamma(t) * \text{Perm}^\gamma(t) \Rightarrow \top$$

Exclusiveness of $\text{Perm}^\gamma(t)$ allows us to define the escrow used to transfer the lock's resource, which contains P and the full single-writer permission of the now-serving counter. This is reflected in the definition of $\mathbf{Escs}(x, \gamma, n)$, which asserts the existence of the escrows for all abstract tickets up to n . Additionally, $\mathbf{Escs}(x, \gamma, n)$ subsumes the proofs that all abstract tickets up to (but excluding) n have been served, due to the fact that the lock's resource can only be stored in exactly one of the escrows at a time, and that the verification maintains that the resource only moves from the escrow of some ticket t to that of $t + 1$. These proofs are needed when a thread updates the now-serving counter $x + \text{ns}$, to release the lock. This is enforced in $\mathbf{NSP}(\gamma)$. When increasing the counter with value $z = n \bmod \mathbf{C}$ to $(n + 1) \bmod \mathbf{C}$, the lock-releasing thread moves the protocol state from n to $n + 1$, and must re-establish $\mathbf{NSP}(\gamma)$ by allocating the escrow for the ticket $n + 1$ of the next lock-acquiring thread.

The ticket counter protocol

Tracking tickets is the main role of the ticket counter. The tracking M is a finite map from participant ids to their corresponding most recent abstract tickets. The domain of M , i.e. the set of ids, is restricted to the set $[0, \dots, \mathbf{C} - 1]$, since there are at most $\mathbf{C}$ threads participating in the lock. If a thread has never obtained any ticket, then it is mapped to -1 . The authoritative part of the map $\bullet M$ is part of $\text{AllTks}^\gamma(M)$, which in turn is stored in the protocol $\text{TCP}(\gamma, x)$ that governs the ticket counter $x + \text{tc}$. The non-authoritative $\circ M$ is split into $\text{MyTkt}^\gamma [i \mapsto M(i)]$'s for $i \in [0, \dots, \mathbf{C} - 1]$. Each $\text{MyTkt}^\gamma [i \mapsto M(i)]$ represents the knowledge of the most recent ticket $M(i)$, as well as the exclusive permission required to acquire a new ticket for the thread i . These permissions are allocated when the lock is created, and then distributed externally to the threads that wish to use the lock.

$\text{TCP}(\gamma, x)$ always keep its authoritative tracker $\text{AllTks}^\gamma(M)$. The maximum abstract ticket of M is $t - 1$, which is also the most recently issued ticket by the ticket counter. Meanwhile, the next ticket to be issued is t , and the current value y of the ticket counter must be the corresponding concrete ticket of t . M of course is injective, since no two threads should get the same ticket. $\text{TCP}(\gamma, x)$ also holds permissions $\star_{j \geq t} \text{Perm}^\gamma(j)$ of to-be-issued abstract tickets. When requesting a ticket, a thread obtains the abstract ticket t (and the concrete ticket y), as well as $\text{Perm}^\gamma(t)$ to use the escrow later. The thread also increases the ticket counter value (concretely and abstractly), and, by doing so, it must show that it has observed the now-serving counter to be at least some state that is not too far away from the ticket counter, which is realized by the conditions between t and n in the definition of TCP . These conditions say that the ticket counter is always ahead of the now-serving counter, but their distance cannot surpass the number of waiting tickets known to the ticket counter, i.e. those that have been issued but are still not reported to the ticket counter as served.

Specification for bounded ticket lock

The specification that we need to verify is very simple, as given in [Figure B.6](#).

$\text{MayAcquire}(x)$ is the permission for a thread to request a ticket and later acquire the lock. The thread must have the right $\text{MyTkt}^\gamma [i \mapsto t]$ to change its most recent ticket t , and it must show that t has been served, by knowing $\boxed{x + \text{ns} : t + 1 \mid \text{NSP}(\gamma)}_R$.

By acquiring the lock, a thread earns $\triangleright P$ and $\text{MayRelease}(x)$, which is the permission to release the lock. $\text{MayRelease}(x)$ includes the writer permission of $\boxed{x + \text{ns} : t \mid \text{NSP}(\gamma)}_W$ for the ticket t that was used to acquire the lock. The thread has to give up this writer permission to update the now-serving counter to $t + 1$, at the same time gaining only the reader permission for $t + 1$, which is enough to re-establish $\text{MayAcquire}(x)$ that can be used for its next turn. Note that P is guarded by a later, since it was put in an escrow. Fortunately there should be always enough steps in

a client program to remove the later.

$$\begin{aligned}
 & \left\{ \triangleright P \right\} \mathbf{newLock}() \left\{ x. \bigstar_{i < \mathbf{C}} \text{MayAcquire}(x) \right\} \\
 & \{ \text{MayAcquire}(x) \} \mathbf{lock}(x) \{ \triangleright P * \text{MayRelease}(x) \} \\
 & \{ \triangleright P * \text{MayRelease}(x) \} \mathbf{unlock}(x) \{ \text{MayAcquire}(x) \}
 \end{aligned}$$

Figure B.6: Specification of bounded ticket locks.

The proof outlines are given Figure B.7, Figure B.8, and Figure B.9. For **newLock**, we only need to set up the default initial states of the protocols, for exactly **C** participants. The proof of **unlock** simply exploit the power of the single writer protocol to release the writer permission while doing the write itself (see §5.2). We explain the proof of **lock** in more detail.

$$\begin{aligned}
 & \{ \triangleright P \} \\
 & \mathbf{let } x = \mathbf{alloc}(2) \mathbf{ in} \\
 & \left\{ \begin{array}{l} \triangleright P * x + \mathbf{ns} \hookrightarrow A * x + \mathbf{tc} \hookrightarrow A \\ \triangleright P * x + \mathbf{ns} \hookrightarrow A * x + \mathbf{tc} \hookrightarrow A \\ * \exists \gamma, M. \bigstar_{t \geq 0} \text{Perm}^\gamma(t) * \text{AllTks}(M_{-1}) * \bigstar_{i < \mathbf{C}} \text{MyTkt}^\gamma[i \mapsto -1] \end{array} \right\} \\
 & x + \mathbf{ns}_{[\mathbf{na}]} := 0; \\
 & \left\{ \begin{array}{l} \boxed{x + \mathbf{ns} : 0 \mid \mathbf{NSP}(\gamma)}_W * x + \mathbf{tc} \hookrightarrow A \\ * \bigstar_{t \geq 0} \text{Perm}^\gamma(t) * \text{AllTks}(M_{-1}) * \bigstar_{i < \mathbf{C}} \text{MyTkt}^\gamma[i \mapsto -1] \end{array} \right\} \\
 & x + \mathbf{tc}_{[\mathbf{na}]} := 0; \\
 & \left\{ \boxed{x + \mathbf{ns} : 0 \mid \mathbf{NSP}(\gamma)}_R * \boxed{x + \mathbf{tc} : \text{Inv} \mid \mathbf{TCP}(\gamma, x)}_R * \bigstar_{i < \mathbf{C}} \text{MyTkt}^\gamma[i \mapsto -1] \right\} \\
 & x \\
 & \left\{ \bigstar_{i < \mathbf{C}} \text{MayAcquire}(x) \right\}
 \end{aligned}$$

Figure B.7: Proof outline for **newLock**

$$\begin{aligned}
 & \{ \triangleright P * \text{MayRelease}(x) \} \\
 & \left\{ \exists \gamma, i, t. \triangleright P * \text{MyTkt}^\gamma[i \mapsto t] * \boxed{x + \mathbf{tc} : \text{Inv} \mid \mathbf{TCP}(\gamma, x)} * \boxed{x + \mathbf{ns} : t \mid \mathbf{NSP}(\gamma)}_W \right\} \\
 & \text{Ctx: } \boxed{x + \mathbf{tc} : \text{Inv} \mid \mathbf{TCP}(\gamma, x)} \\
 & \mathbf{let } z = x + \mathbf{ns}_{[\mathbf{at}]} \mathbf{ in} \\
 & \left\{ z = t \bmod \mathbf{C} * \triangleright P * \text{MyTkt}^\gamma[i \mapsto t] * \boxed{x + \mathbf{ns} : t \mid \mathbf{NSP}(\gamma)}_W \right\} \\
 & x + \mathbf{ns}_{[\mathbf{at}]} := (z + 1) \bmod \mathbf{C} \\
 & \left\{ \text{MyTkt}^\gamma[i \mapsto t] * \boxed{x + \mathbf{ns} : t + 1 \mid \mathbf{NSP}(\gamma)}_R \right\} \\
 & \{ \text{MayAcquire}(x) \}
 \end{aligned}$$

Figure B.8: Proof outline for **unlock**

B.5 PROOF STEPS FOR LOCK

The proof outline for the **lock** procedure is given in Figure B.9. We explain the main reasoning below.

Figure B.9: Proof outline for **lock**

$$\begin{array}{l}
\{ \text{MayAcquire}(x) \} \\
\left\{ \begin{array}{l} \exists \gamma, i, t_i. \text{MyTkt}^\gamma [i \mapsto t_i] * \boxed{x + \text{tc} : \text{Inv} \mid \text{TCP}(\gamma, x)}_R \\ * t_i \neq -1 \Rightarrow \boxed{x + \text{ns} : t_i + 1 \mid \text{NSP}(\gamma)}_R \end{array} \right\} \\
\text{let } y = \text{FAI}(x + \text{tc}) \text{ in} \\
\left\{ \begin{array}{l} \exists t, n_0. y = t \bmod \mathbf{C} \wedge t < n_0 + \mathbf{C} * \text{MyTkt}^\gamma [i \mapsto t] \\ * \text{Perm}^\gamma(t) * \triangleright \boxed{x + \text{ns} : n_0 \mid \text{NSP}(\gamma)}_R \end{array} \right\} \\
\text{Ctx: } y = t \bmod \mathbf{C} \wedge t < n_0 + \mathbf{C} \\
\text{Ctx: } \boxed{x + \text{tc} : \text{Inv} \mid \text{TCP}(\gamma, x)} \\
\text{repeat} \\
\left\{ \begin{array}{l} \exists n \geq n_0. \text{MyTkt}^\gamma [i \mapsto t] * \text{Perm}^\gamma(t) * \boxed{x + \text{ns} : n \mid \text{NSP}(\gamma)}_R \end{array} \right\} \\
\text{let } z = x + \text{ns}_{[\text{at}]} \text{ in} \\
\left\{ \begin{array}{l} \exists n' \geq n. z = n' \bmod \mathbf{C} * \Box \text{Escs}(x, \gamma, n') \\ * \text{MyTkt}^\gamma [i \mapsto t] * \text{Perm}^\gamma(t) * \boxed{x + \text{ns} : n' \mid \text{NSP}(\gamma)}_R \end{array} \right\} \\
y = z \\
\left\{ \begin{array}{l} \exists n \geq n_0. y = z * z = n \bmod \mathbf{C} * \Box \text{Escs}(x, \gamma, n) \\ * \text{MyTkt}^\gamma [i \mapsto t] * \text{Perm}^\gamma(t) * \boxed{x + \text{ns} : n \mid \text{NSP}(\gamma)}_R \end{array} \right\} \\
\text{Have } n \leq t < n_0 + \mathbf{C} \leq n + \mathbf{C} \text{ and } t \bmod \mathbf{C} = n \bmod \mathbf{C} \\
\text{Ctx: } t = n \\
\left\{ \begin{array}{l} \text{MyTkt}^\gamma [i \mapsto t] * \text{Perm}^\gamma(t) * \left[\text{Perm}^\gamma(t) \rightsquigarrow P * \boxed{x + \text{ns} : t \mid \text{NSP}(\gamma)}_W \right] \\ \text{MyTkt}^\gamma [i \mapsto t] * \triangleright P * \triangleright \boxed{x + \text{ns} : t \mid \text{NSP}(\gamma)}_W * \boxed{x + \text{ns} : t \mid \text{NSP}(\gamma)}_R \\ \triangleright P * \text{MyTkt}^\gamma [i \mapsto t] * \boxed{x + \text{ns} : t \mid \text{NSP}(\gamma)}_W \end{array} \right\} \\
\{ \triangleright P * \text{MayRelease}(x) \}
\end{array}$$

- With the permission $\text{MyTkt}^\gamma [i \mapsto t_i]$, a thread i acquires a new abstract ticket t (concretely y) along with the permission $\text{Perm}^\gamma(t)$, and updates the tracker for i to t .

It also increases the ticket counter concretely to $(y + 1) \bmod \mathbf{C}$ along with an abstract ticket $t + 1$, by which the ticket counter demands $\boxed{x + \text{ns} : n_0 \mid \text{NSP}(\gamma)}_R$ for some good enough n_0 . This demand is fulfilled either by what the thread has locally, or what is already in the interpretation previously. In either way the thread updates its observation of $\text{NSP}(\gamma)$ to n_0 .

- By entering the loop, the thread keeps updating its observation of $\text{NSP}(\gamma)$ to some state n later than n_0 . When the loop exits, the thread knows that the two concrete tickets y and z are the same, and this fact should imply that their corresponding abstract tickets t and n are also the same. To show this we need to use a subtle reasoning with the single writer protocol.

Since we already have $t \bmod \mathbf{C} = n \bmod \mathbf{C}$ and $t < n + \mathbf{C}$ by acquiring the ticket and loop-checking the counter, we only need to show $n \leq t$ to show $n = t$. This matches the intuition that the now-serving counter cannot surpass the unserved ticket, since we have locally the permission $\text{Perm}^\gamma(t)$. We assume the contrary $t < n$ and derive a contradiction. Note that by reading the now-serving counter we also obtain $\text{Escs}(x, \gamma, n)$. Then we can use $\text{Perm}^\gamma(t)$ with the escrow for t

to obtain $\triangleright \boxed{x + \text{ns} : t \mid \mathbf{NSP}(\gamma)}_W$. Combining with our locally-known $\boxed{x + \text{ns} : n \mid \mathbf{NSP}(\gamma)}_R$, it must be that $n \leq t$: *contradiction!*

- Knowing that $n = t$, we use the escrow with $\text{Perm}^\gamma(t)$ to acquire the lock, and gain $\triangleright P$. We also gain $\triangleright \boxed{x + \text{ns} : t \mid \mathbf{NSP}(\gamma)}_W$, which we combine with $\boxed{x + \text{ns} : t \mid \mathbf{NSP}(\gamma)}_R$ to remove the later, and complete the $\text{MayRelease}(x)$ permission.

C

iGPS Protocol Model

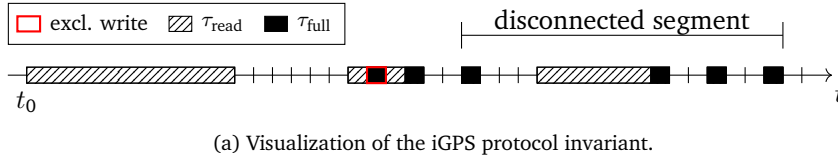


Figure C.1: The iGPS protocol invariant.

$$\begin{aligned}
 \text{PrInv}_\tau(\ell) &\triangleq \exists \Delta, \delta_x \in \Delta. \\
 &\text{Hist}(\ell, \{(v, t, V) \mid (s, v, t, V) \in \Delta\}) \\
 &* [\bullet \bullet \Delta \bullet \circ \Delta] : \text{LH} * [\bullet \delta_x : \text{XW}] \\
 &* \left(*_{(s, v, _, V) \in \Delta} \tau_{\text{read}}(s, v)(V) \right) \\
 &* \left(*_{(s, v, t, V) \in \Delta} (\nexists \delta \in \Delta. \delta.t = t + 1) \Rightarrow \tau_{\text{full}}(s, v)(V) \right) \\
 &* \text{Consistent}(\Delta) * \text{Unique}(\Delta) * \text{FinalInv}(\Delta)
 \end{aligned}$$

(b) Definition of the iGPS protocol invariant.

The model of iGPS protocols consists of two parts: a global protocol invariant $\text{PrInv}_\tau(\ell)$ shared by all protocol variants, and local protocol assertions given out to clients, which depend on the particular protocol variant. We will now explain both parts in detail. The definitions are given in [Figure C.1b](#).¹

The protocol invariant owns, among other resources, the location's history $\text{Hist}(\ell, h)$ – though h will be expressed differently, as we explain below. In addition to the physical history, the invariant also owns the authoritative fragment of the *logical history* Δ – an augmented history that write events paired with protocol states.

The protocol invariant further owns the protocol interpretation for every state in Δ . As all states are potentially targets of reads, we store τ_{read} for every state in the history. To enable ownership transfer via CAS, we additionally need to store τ_{full} for those states whose associated write event can still be target of an update operation.

This condition is sufficient for non-exclusive protocols. However, single-writer protocols offer an additional mechanism of transferring out resources from the protocol interpretation: exclusive writes. To support

¹ The definitions differ slightly from the Rocq mechanization to simplify the presentation.

both protocol variants, the invariant tracks the most recent exclusive write (s_x, v_x, V_x) and stores τ_{full} for δ_x as well as any later write that can be target of CAS. In the case of non-exclusive protocols, δ_x is set to the initial protocol write and all CAS-able writes maintain their full interpretation τ_{full} .

We track δ_x with an additional authoritative piece of ghost state. The protocol invariant holds the authoritative fragment $\bullet \Delta_x$ while the non-authoritative fragment $\circ \Delta_x$ is given to the writer – or put into the shared invariant PrShared_τ in the case of non-exclusive protocols.

Apart from those resources, the protocol invariant imposes three conditions on the physical history h and the logical history Δ . The most important one is *consistency*: the modification order on h must correspond to the protocol order on Δ . The *Consistent* assertion captures exactly this property.

Additionally, we need to make sure that the logical history does not assign multiple protocol states to a write event. To this end, *Unique* asserts that Δ contains any write event at most once. With this restriction in place, we can now express h in terms of Δ by forgetting the protocol states in Δ , i.e. $h = \{(v, t, V) \mid (s, v, t, V) \in \Delta\}$.

Finally, we need an additional piece of information on the logical history: Δ contains only final states, i.e. states reachable from any other state, in the *disconnected segment*. We call this property *FinalInv*. The disconnected segment – illustrated in [Figure C.1a](#) – spans all writes which are not connected to the most recent exclusive write. The write events contained in that segment share an important property: they could be *mo*-after newly inserted writes (which are always *mo*-after the most recent exclusive write). To re-establish *Consistent* for new writes, we need to be able to relate their protocol state to any existing *mo*-later write. *FinalInv* guarantees that the state associated with the newly inserted write comes before any state in the disconnected segment.

Before we turn to the local protocol assertions, we define the PCMs used in PrInv_τ . The PCM for the most recent exclusive write δ_x is a simple *AUTH* construction similar to the ones shown in §4, providing both an authoritative fragment for PrInv_τ as well as an exclusive non-authoritative fragment to be given to the writer or to PrShared_τ . The PCM for logical histories is more involved, as it needs to provide three kinds of fragments: (1) an authoritative fragment held by PrInv_τ , (2) an exclusive fragment given to the writer or to PrShared_τ , representing the right to change the logical history – and thus also the physical one – and (3) a persistent fragment used in the (reader) protocol assertion, representing a lower bound on protocol state. To fulfill these requirements, we construct the rather intimidating double-AUTH PCM LH below.

$$\begin{aligned}
 \text{LE} &\triangleq \text{State} \times \text{Val} \times \text{Time} \times \text{View} && \text{(logical event)} \\
 \text{LES} &\triangleq \mathcal{P}_\cup(\text{LE}) && \text{(logical event set)} \\
 \text{XW} &\triangleq \text{AUTH}(\text{EX}(\text{LE})) && \text{(most recent excl. event)} \\
 \text{LH} &\triangleq \text{AUTH}(\text{AUTH}(\text{LES})) && \text{(auth. logical event set)}
 \end{aligned}$$

$$\begin{aligned}
\text{Reader}(\delta) &\triangleq [\circ(\circ\{\delta\}) : \text{LH}] \\
\text{Writer}(\delta) &\triangleq \exists \Delta. [\circ(\bullet \Delta \cdot \circ \Delta) : \text{LH}] * \delta \in \Delta * \forall \delta' \in \Delta. \delta'.t \leq \delta.t \\
\text{PrShared}_\tau(t_x) &\triangleq \exists \delta, \delta_x. \text{Writer}(\delta) * [\circ \delta_x : \text{XW}] * \delta_x.t = t_x. \\
\boxed{\ell : s \mid \tau} &\triangleq \lambda V. [\text{PrInv}_\tau(\ell)] * \exists V_R \sqsubseteq V, t_x \leq V(\ell). \\
&\quad \text{Reader}((s, -, V_R)) * [\text{PrShared}_\tau(t_x)] \\
\boxed{\ell : s \mid \tau}_R &\triangleq \lambda V. [\text{PrInv}_\tau(\ell)] * \exists V_R \sqsubseteq V. \text{Reader}((s, -, V_R)) \\
\boxed{\ell : s \mid \tau}_W &\triangleq \lambda V. [\text{PrInv}_\tau(\ell)] * \exists V_W \sqsubseteq V, v_w, t_w. \\
&\quad \text{Writer}((s, v_w, t_w, V_W)) * [(s, v_w, t_w, V_W) : \text{XW}]
\end{aligned}$$

Figure C.2: Definition of the iGPS protocol assertions for non-exclusive protocols and single-writer protocols.

With these definitions we can now define protocol assertions, which are presented in [Figure C.2](#). As before, we elide ghost and invariant names, which add some bookkeeping overhead to the definitions.

Bibliography

- [AM16] Tatsuya Abe and Toshiyuki Maeda. “Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models”. In: *APLAS*. 2016, pp. 63–84 (cit. on p. 53).
- [Bat+15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. “The Problem of Programming Language Concurrency Semantics”. In: *ESOP*. 2015 (cit. on pp. 25, 55).
- [Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ concurrency”. In: *POPL*. ACM, 2011, pp. 55–66 (cit. on p. 21).
- [BAP15] Richard Bornat, Jade Alglave, and Matthew J. Parkinson. “New Lace and Arsenic: adventures in weak memory with a program logic”. In: *CoRR* abs/1512.01416 (2015) (cit. on p. 53).
- [dDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. “TaDA: A Logic for Time and Data Abstraction”. In: *ECOOP*. 2014 (cit. on pp. 19, 20).
- [Din+10] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. “Concurrent abstract predicates”. In: *ECOOP 2010*. Vol. 6183. LNCS. Springer, 2010, pp. 504–528 (cit. on p. 19).
- [DV16] Marko Doko and Viktor Vafeiadis. “A Program Logic for C11 Memory Fences”. In: *VMCAI*. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 413–430 (cit. on pp. 19, 53).
- [DV17] Marko Doko and Viktor Vafeiadis. “Tackling Real-Life Relaxed Concurrency with FSL++”. In: *ESOP*. 2017 (cit. on p. 53).
- [Dre] Derek Dreyer. “The RustBelt project”. <http://plv.mpi-sws.org/rustbelt/> (cit. on p. 20).
- [Fen09] Xinyu Feng. “Local rely-guarantee reasoning”. In: *POPL*. 2009, pp. 315–327 (cit. on p. 19).
- [Jon83] C. B. Jones. “Tentative steps toward a development method for interfering programs”. In: *TOPLAS* 5.4 (1983), pp. 596–619 (cit. on p. 20).
- [Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. “Higher-order ghost state”. In: *ICFP*. 2016, pp. 256–269 (cit. on pp. 19, 20, 33, 49).

- [Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning”. In: *POPL*. 2015, pp. 637–650 (cit. on pp. 19, 20, 35).
- [Kan+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. “A promising semantics for relaxed-memory concurrency”. In: *POPL*. 2017, pp. 175–189 (cit. on p. 55).
- [Kre+17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. “The Essence of Higher-Order Concurrent Separation Logic”. In: *ESOP*. 2017 (cit. on pp. 19, 20, 34).
- [KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *POPL*. ACM, 2017, pp. 205–217 (cit. on pp. 19, 20, 50, 53).
- [KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. “A relational model of types-and-effects in higher-order concurrent separation logic”. In: *POPL*. 2017, pp. 218–231 (cit. on p. 53).
- [LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. “Taming Release-acquire Consistency”. In: *POPL*. POPL 2016. ACM, 2016, pp. 649–662. ISBN: 978-1-4503-3549-2 (cit. on pp. 22, 54, 55).
- [LV15] Ori Lahav and Viktor Vafeiadis. “Owicki-Gries Reasoning for Weak Memory Models”. In: *Automata, Languages, and Programming, ICALP 2015*. Vol. 9135. LNCS. Springer, 2015, pp. 311–323 (cit. on pp. 19, 53).
- [Lam79] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691 (cit. on p. 21).
- [MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *VMCAI*. 2016, pp. 41–62 (cit. on p. 53).
- [Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. “Communicating State Transition Systems for Fine-Grained Concurrent Resources”. In: *ESOP*. 2014 (cit. on pp. 19, 42).
- [OHe07] Peter W. O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307 (cit. on p. 19).
- [Rey02] John C. Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *LICS*. 2002 (cit. on p. 19).

- [Rid10] Tom Ridge. “A rely-guarantee proof system for x86-TSO”. In: *VSTTE 2010*. Vol. 6217. LNCS. Springer, 2010, pp. 55–70 (cit. on p. 53).
- [SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. “Mechanized verification of fine-grained concurrent programs”. In: *PLDI*. 2015, pp. 77–87 (cit. on p. 20).
- [Sie+15] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. “A Separation Logic for Fictional Sequential Consistency”. In: *ESOP 2015*. Vol. 9032. LNCS. Springer, 2015, pp. 736–761 (cit. on p. 53).
- [Sum17] Alexander Summers. Personal communication. 2017 (cit. on p. 53).
- [SB14] Kasper Svendsen and Lars Birkedal. “Impredicative Concurrent Abstract Predicates”. In: *ESOP*. 2014 (cit. on pp. 19, 20, 42).
- [TDV15] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. “Verifying Read-copy-update in a Logic for Weak Memory”. In: *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. ACM, 2015, pp. 110–120 (cit. on pp. 41, 46, 51, 71).
- [TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”. In: *ICFP*. ACM, 2013 (cit. on pp. 19, 20, 42).
- [TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”. In: *OOPSLA*. OOPSLA 2014. ACM, 2014, pp. 691–707 (cit. on pp. 19, 21, 23, 41, 46, 53).
- [VN13] Viktor Vafeiadis and Chinmay Narayan. “Relaxed Separation Logic: A program logic for C11 concurrency”. In: *OOPSLA 2013*. ACM, 2013, pp. 867–884 (cit. on pp. 19, 21, 23, 41, 49, 53).
- [VP07] Viktor Vafeiadis and Matthew Parkinson. “A marriage of rely/guarantee and separation logic”. In: *CONCUR 2007*. Vol. 4703. LNCS. Springer, 2007, pp. 256–271 (cit. on p. 19).

Part II

Mtac2: Typed Tactics for Backward Reasoning in Rocq

Rocq supports a range of built-in tactics, which are engineered primarily to support *backward reasoning*. Starting from a desired goal, the Rocq programmer can use these tactics to manipulate the proof state interactively, applying axioms or lemmas to break the goal into subgoals until all subgoals have been solved. Additionally, it provides support for *tactic programming* via OCaml and Ltac, so that users can roll their own custom proof automation routines.

Unfortunately, though, these tactic languages share a significant weakness. They do not offer the tactic programmer any static guarantees about the soundness of their custom tactics, making large tactic developments difficult to maintain. To address this limitation, Ziliani *et al.* [Zil+15] previously proposed **Mtac**, a new typed approach to custom proof automation in Rocq which provides the static guarantees that OCaml and Ltac are missing. However, despite its name, Mtac is really more of a metaprogramming language than a full-blown tactic language: it lacks an essential feature of tactic programming, namely the ability to directly manipulate Rocq’s proof state and perform backward reasoning on it.

We present **Mtac2**, a next-generation version of Mtac that combines its support for typed metaprogramming with additional support for the programming of backward-reasoning tactics in the style of Ltac. In so doing, Mtac2 introduces a novel feature in tactic programming languages—what we call *typed backward reasoning*. With this feature, Mtac2 is capable of statically ruling out several classes of errors that would otherwise remain undetected at tactic definition time. We demonstrate the utility of Mtac2’s typed tactics by porting several tactics from a large Rocq development, the Iris Proof Mode, from Ltac to Mtac2.

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

8

Introduction

The Rocq proof assistant provides a rich dependently-typed framework in which to formalize mathematics and programming language metatheory. Although Rocq proofs ultimately compile down to proof terms in the language of Type Theory, it is not practical for Rocq programmers to write such proof terms manually. Instead, to make proof construction feasible, Rocq supports a range of built-in *tactics*, which are engineered primarily to support *backward reasoning*. Starting from a desired goal, the Rocq programmer can use these tactics to manipulate the proof state interactively, applying axioms or lemmas to break the goal into subgoals until all subgoals have been solved. This style of backward reasoning is convenient because Rocq can often infer many details about how to instantiate lemmas by inspection of the current proof state, and as a result the Rocq user can omit these tedious details from their interactive proof scripts.

In addition to built-in tactics, Rocq supports two *tactic languages*—OCaml and Ltac—that enable users to roll their own tactics. With these languages, Rocq users can build custom proof automation routines, which simplify proof scripts further by exploiting domain-specific knowledge about the structure of proofs. OCaml is the language used to implement Rocq itself, and the main benefit of using it is performance: tactics written in OCaml are compiled rather than interpreted, and can make use of imperative data structures for efficiency. That said, it is not ideal for general-purpose tactic programming because it exposes developers to low-level and potentially unsafe details of Rocq’s internals, *e.g.*, forcing them to work directly with de Bruijn representations of terms. In contrast, Ltac is a higher-level dynamic language that allows for rapid prototyping of tactics directly within the Rocq environment. Ltac is an interpreted language, and thus less efficient than OCaml, but it provides a more convenient representation of proof terms using the concrete syntax of Rocq.

Unfortunately, though, both OCaml and Ltac share a common, and rather significant, weakness. Neither offers the tactic programmer any static guarantees about the soundness of their custom tactics: it is easy to define tactics that are accepted by the OCaml compiler (or Ltac interpreter) but that construct ill-typed terms or generate inscrutable errors

[Zil+13] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

when they are applied.

To address this limitation, Ziliani *et al.* [Zil+13; Zil+15] proposed **Mtac**, a new interpreted language for custom proof automation in Rocq which provides the static guarantees that OCaml and Ltac are missing. Mtac is based on the key realization that, at its core, tactic programming is essentially functional programming—of the sort already supported by Rocq’s functional language Gallina—extended with certain *effects*, such as general recursion, syntax inspection, and exception handling. Although these effectful operations are not directly available in Gallina, their static semantics can be described using the *type structure* of Gallina, and they can thus be supported by means of a monadic extension to Gallina—not unlike the way Haskell extends its pure functional core with computational effects via the IO monad. Specifically, Mtac extends Rocq with a monadic type $\mathbb{M} \ A$, representing the type of Mtac tactics that, when applied, may diverge or fail, but if they terminate successfully, will produce a Rocq term of type A .

However, despite its name, Mtac is really more of a metaprogramming language than it is a full-blown tactic language. In particular, compared to OCaml and Ltac, it lacks an essential feature of tactic programming, namely the ability to directly manipulate Rocq’s proof state and perform backward reasoning on it.

In this chapter, we present **Mtac2**, a next-generation version of Mtac that combines its support for typed metaprogramming with additional support for the programming of backward-reasoning tactics in the style of Ltac. In so doing, Mtac2 introduces a novel feature in tactic programming languages—what we call *typed backward reasoning*.

8.1 AN EXAMPLE MOTIVATING TYPED BACKWARD REASONING

To explain and motivate typed backward reasoning more concretely, we now turn to a real example of tactic programming taken from Iris Proof Mode (IPM) [KTB17]—a tactic library for supporting interactive development of separation-logic proofs in Rocq. It has been used for a variety of applications in advanced program verification, including a safety proof for a realistic subset of the Rust programming language [Jun+18a; KTB17; Tim+18; Kai+17; SGD17].

To give a brief impression of IPM, we provide an example proof state of a program verification performed in Iris. The program in question adds the number 4 to itself, then adds the result to the contents of location x , which we know to contain 8. The program’s postcondition guarantees that the resulting value is 16 and that ownership of x (still with value 8) is returned.

```

1 "H" : x ↦ 8
2 -----*
3 WP let: "t" := 4 + 4 in
4   !x + "t" {v, v = 16 * x ↦ 8 }
```

[KTB17] Krebbers et al., “Interactive Proofs in Higher-Order Concurrent Separation Logic”

[Jun+18a] Jung et al., “RustBelt: Securing the foundations of the Rust programming language”

[Tim+18] Timany et al., “A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST”

[Kai+17] Kaiser et al., “Strong logic for weak memory: Reasoning about release-acquire consistency in Iris”

[SGD17] Swasey et al., “Robust and compositional verification of object capability patterns”

To account for separation logic hypotheses (as opposed to Rocq hypotheses), the proof mode introduces a new context (delimited by `----`), called the *spatial* context.¹ This spatial context contains assertions describing ownership of resources, such as the *points-to* assertion $x \mapsto 8$, which asserts ownership of the location x storing the value 8.

The remaining part of the proof state is the IPM goal. In our case, we are proving an Iris weakest precondition, *i.e.*, a proof of partial program correctness. The syntax used here is `WP e {{ v,  $\Phi$  v }}`, which denotes the weakest precondition implying that expression e is safe to execute and that its resulting value v (should it terminate) satisfies the postcondition Φ .

IPM offers a variety of tactics to manage separation logic contexts, to handle the various separation logic constructs (separating conjunction, magic wand, higher-order quantification, etc.), and to apply existing separation logic lemmas and theorems. Moreover, it also offers a set of tactics to perform symbolic execution. It is one of these symbolic execution tactics that we focus on in our demonstration: `wp_pure`, a tactic to symbolically execute pure program steps such as beta reductions and arithmetic operations.

In our program above, the next step of computation is an addition of two integers. To instruct IPM to symbolically execute the operation, we invoke `wp_pure ( _ + _ )`.² The proof mode uses type classes to find a sufficient condition for executing the operation, as well as to find the result of the reduction. A sufficient condition for division, for example, could be that the denominator is not 0. For the addition operation in our example, this condition is trivial, *i.e.*, `True`, and hence automatically solved. The remaining program, now containing the result of the addition, is left as an open goal for the user to prove:

```
1 "H" : x ↦ 8
2 -----*
3 WP let: "t" := 8 in
4 !x + "t" {{ v, v = 16 * x ↦ 8 }}
```

At the core of the symbolic execution infrastructure lie a set of lemmas that justify single steps of symbolic execution. As an example, consider the lemma `tac_wp_pure` in Figure 8.1, which serves as an umbrella lemma for symbolically executing pure execution steps. To understand the type of `tac_wp_pure`, we need to take a peek behind the sugary syntax curtains. First of all, the proof states shown above are merely syntactic sugar for the underlying entailment relation $\text{envs_entails } \Delta P$. Here, Δ is the separation logic context, and P the IPM conclusion (or IPM goal). Secondly, the weakest precondition assertions shown in the example above (written `WP e @ s ; E {{  $\Phi$  }}` in its full form) is syntactic sugar for `wp s E e  $\Phi$` . The arguments s and E allow Iris to express more general notions of weakest precondition and can be ignored for the purpose of this chapter.

The first hypothesis of `tac_wp_pure` is a type class, `PureExec`, which proves that the proposition φ is a sufficient condition for e_1 to reduce to e_2 . Note that in this type class search problem, e_1 is taken as input—

¹ In fact, IPM proofs also involve a “persistent” context for freely duplicable separation logic hypotheses. For simplicity, we ignore the persistent context in this chapter.

² Note that it is possible to give a wildcard expression, `_`, to `wp_pure` in which case the tactic will symbolically execute the first pure expression it finds. It is, however, sometimes useful for proof maintainability to explicitly specify the shape of the expression to be executed.

it is derived from the current goal—and both φ and $e2$ are considered outputs. For example, `PureExec (y≠0) (#x div #y) (#(x div y))` is a proof that the integer division of x by y reduces if the denominator is not 0, and that the result of the reduction is $\#(x \text{ div } y)$, where `div` is now the meta-level integer division function. (The `#` notation denotes integer literals in the expression language.) Thus, once we have established `PureExec  $\varphi$  e1 e2`, to prove `envs_entails  $\Delta$  (WP e1 {{  $\Phi$  }})`, it suffices to show φ and `envs_entails  $\Delta$  (WP e2 {{  $\Phi$  }})`.

IPM provides wrapper tactics for lemmas like `tac_wp_pure`. These tactics perform the necessary pre- and post-processing and error reporting in case of failure. The Ltac tactic wrapping `tac_wp_pure`, called `wp_pure`, is presented in Figure 8.1 (from Line 7 onward).

The tactic first checks that the IPM goal matches the weakest precondition assertion. If that is the case, it attempts to find an occurrence of the given argument (`epat`) in evaluation position and apply `tac_wp_pure`.

Figure 8.1: An IPM lemma, `tac_wp_pure`, describing the effect of symbolically executing a pure expression; and the Ltac code for `wp_pure`, a tactic to symbolically execute pure reduction steps.

```

1 Lemma tac_wp_pure `{{heapG  $\Sigma$ }}  $\Delta$  s E e1 e2  $\varphi$   $\Phi$  :
2   PureExec  $\varphi$  e1 e2 ->
3    $\varphi$  ->
4   envs_entails  $\Delta$  (wp s E e2  $\Phi$ ) ->
5   envs_entails  $\Delta$  (wp s E e1  $\Phi$ ).
6
7 Tactic Notation "wp_pure" open_constr(epat) :=
8   lazymatch goal with
9   | |- envs_entails _ (wp ?s ?E ?e ? $\Phi$ ) =>
10    let e := eval simpl in e in
11    reshape_expr e ltac:(fun K e' =>
12      unify e' epat;
13      eapply (tac_wp_pure _ _ _ (fill K e'));
14      [apply _ (* PureExec *)
15       |try fast_done (* The pure condition for PureExec *)
16        |wp_expr_simpl_subst; try wp_value_head (* new goal *)
17      ])
18    || fail "wp_pure: cannot find" epat "in" e
19       "or" epat "is not a reduct"
20    | _ => fail "wp_pure: not a 'wp'"
21 end.
```

The search for expressions in evaluation position is taken care of by the helper tactic `reshape_expr`. To understand this tactic, note that the language used in the example is based on *evaluation contexts*. To find expressions in evaluation position, it thus suffices to decompose the expression e into evaluation contexts K and the corresponding redexes e' , such that K and e' can be “plugged” together to yield e . This plugging operation is called `fill` in Iris. We give more details on evaluation contexts in Section 12.

The job of `reshape_expr` is to enumerate all possible decompositions of e into K and e' and call its continuation with each decomposition until the continuation succeeds. In our example, the continuation—the remainder of `wp_pure`—makes sure to filter out any redexes e' which do not match the given pattern `epat`. Once a suitable candidate has been found, `wp_pure` applies `tac_wp_pure`. The various subgoals are either dispatched by type

class search (`apply _`), treated in a best effort way (`try fast_done` for the proof of φ), or—in the case of the last goal—carefully massaged by simplifying the remaining expression and by checking if we have reached the end of our program, in which case only the postcondition remains to be proven.

The tactic as written above is probably correct. However, Ltac is not helping the developer to enforce that. To see that, let us look at various ways that writing the tactic could have gone wrong:

1. With Ltac’s `lazymatch` we ensure that the shape of the goal matches the conclusion of `tac_wp_pure`. However, Ltac does not guarantee that the lemma we give to `eapply` matches the goal specified in the `lazymatch` branch and we could have tried to apply a completely different lemma. We should be able to enforce that the conclusion of the lemma given to `eapply` indeed matches the shape of the goal.
2. Similarly, Ltac does not even check if our invocation of `tac_wp_pure` is well-formed, *i.e.*, that its arguments are of the correct type.
3. Additionally, Ltac guarantees neither that follow-up tactics can be applied to their respective subgoals nor that we supplied a reasonable number of follow-up tactics (*i.e.*, not too many).
4. Lastly, Ltac offers very limited error raising and handling facilities. The mechanism is not based on ML-like exceptions but instead on the concept of *failures at level n* . Handlers such as the `try` combinator catch errors at level 0 and otherwise decrease the level and propagate the error. This makes it difficult to write modular error handling code that distinguishes intended forms of failure—*e.g.*, for driving the backtracking in `reshape_expr`—from unintended forms of failure.

In short, any change to the various pieces involved in `wp_pure`—IPM definitions, Iris assertions such as `WP`, follow-up tactics, `tac_wp_pure`, and other supporting lemmas—may lead to runtime failures that could have easily been prevented if tactics supported *static typing*.

Using Mtac2 we can write such statically-typed tactics, and in Section 12, we present an Mtac2 version of `wp_pure` which addresses all the shortcomings of the Ltac version:

1. It statically ensures that the conclusion of `tac_wp_pure` applied to the given arguments matches the current goal.
2. It statically ensures that each of the arguments given to `tac_wp_pure` has the right type.
3. It statically ensures that all remaining arguments (*i.e.*, hypotheses) to `tac_wp_pure` are provided for by the follow-up tactics, *and* that each of these follow-up tactics can solve its respective subgoal.
4. It uses exceptions to drive the backtracking in `reshape_expr` and thus avoids any confusion between backtracking and actual failure.

8.2 CONTRIBUTIONS

Our contributions are the following.

- We develop Mtac2, an extension of Mtac to support backward reasoning. In particular, we introduce dynamically-typed backward reasoning in the style of Ltac (Section 10). Unlike in Ltac, *all* Mtac2 tactics are defined in Rocq itself.
- We further introduce the concept of *typed tactics for backward reasoning*, which follows naturally from the combination of Rocq’s rich type system and Mtac2’s tactic infrastructure (Section 11). With typed tactics, Mtac2 is capable of statically ruling out several classes of errors that would otherwise remain undetected at tactic definition time.
- We demonstrate the utility of Mtac2’s typed tactics by porting several tactics from the Rocq development of Iris Proof Mode from Ltac to Mtac2 (Section 12).
- We substantially modify the Mtac language to support several missing features and to overcome difficulties that arose during the construction of realistic tactics (Section 13 and Section 14).

We begin with a review of the basics of Mtac (Section 9) and conclude with a comparison to related work (Section 15). Mtac2, along with all the examples in this chapter, is available online [Kai+18].

[Kai+18] Kaiser et al., “Coq repository for Mtac2”

9

The core of Mtac by example

In this section, we provide an overview of the basic concepts of the Mtac framework that are shared between the original Mtac1 (*i.e.*, the Mtac implementation as described by [Zil+15]) and our new implementation of Mtac2 as described in this thesis. This is largely review, since most of these concepts are inherited from the original Mtac1. However, they form the essential building blocks of Mtac2’s tactic infrastructure, which we will introduce in the sections that follow. As we proceed, we will note (with footnotes) any points where Mtac2 diverges from Mtac1.

Mtac is a framework for typed metaprogramming in Rocq. Mtac takes an unusual approach to metaprogramming in that it represents metaprograms¹ using the language of Rocq itself, enlisting Rocq’s strong type system to ensure the framework’s primary guarantee: If a well-typed metaprogram produces a value, that value will be well-typed.²

Mtac’s primitives are members of a type family, denoted M , that forms a monad. The types and syntax of `bind` and `ret` are given below. The second syntax for `bind`, `('(x) <- ta; tb)`, combines `bind` and irrefutable pattern matching on the result into a single syntactic construct. It uses Rocq’s built-in syntax for irrefutable pattern matching on function arguments written `(fun '(x) => tb)`. As an example, we write `('(x,y) <- ta; tb)` to deconstruct a pair.

```
bind: forall {X Y : Type}, M X -> (X -> M Y) -> M Y.  
ret: forall {X : Type}, X -> M X.
```

```
x <- ta; tb ≜ bind ta (fun x => tb)  
'(x) <- ta; tb ≜ bind ta (fun '(x) => tb)  
ta >=> fb ≜ bind ta fb
```

Mtac uses this monad to encapsulate non-termination and the possibility of failure during the execution of metaprograms. To deal with failure, Mtac offers an exception mechanism akin to that of many languages in the ML family. We can thus express Mtac’s guarantee more precisely: Executing a metaprogram of type $M\ T$ will either diverge, raise an (uncaught) exception, or produce a value of type T .³

Apart from the monadic operations, M provides a broad variety of useful primitives, allowing users to call various unification algorithms; inspect, destruct, and construct terms; etc. We illustrate the various primitives of

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

¹ A superficial change—but important for the sake of clarity—between Mtac1 and Mtac2 lies in the terminology: every Mtac1 program was referred to as a tactic (actually, an Mtactic) while in Mtac2, tactics are a subset of Mtac2 programs, which we generally now call “metaprograms”.

² Modulo the guard condition on fixed-points appearing in the produced value, which is not statically guaranteed to hold by Mtac metaprograms.

³ For a more formal account of this guarantee, which differs slightly in detail between Mtac1 and Mtac2, see Section 13.

Mtac by developing a simple, incomplete first-order tautology solver. For simplicity, the solver only focuses on the introduction of connectives and does not attempt to eliminate connectives in hypotheses. The code of the solver is given in Figure 9.1.

Figure 9.1: A limited first-order tautology solver in Mtac1.

```

1 Definition solve_tauto : forall (l : list dyn) (P : Prop), M P :=
2   mfix2 f (l : list dyn) (P : Prop) : M P :=
3     mtry lookup P l
4     with NotFound =>
5       mmatch P as P' return M P' with
6       | True => ret I
7       | [? Q1 Q2] Q1 ∧ Q2 =>
8         q1 <- f l Q1;
9         q2 <- f l Q2;
10        ret (conj q1 q2)
11      | [? Q1 Q2] Q1 ∨ Q2 =>
12        mtry
13          q1 <- f l Q1;
14          ret (or_introl q1)
15        with TautoFail =>
16          q2 <- f l Q2;
17          ret (or_intror q2)
18        end
19      | [? (Q1 Q2 : Prop)] Q1 -> Q2 =>
20        ν q1,
21        q2 <- f (Dyn q1 :: l) Q2;
22        abs_fun q1 q2
23      | [? X (Q : X -> Prop)] (exists x : X, Q x) =>
24        x <- evar X;
25        q <- f l (Q x);
26        b <- is_evar x;
27        if b then
28          raise TautoFail
29        else
30          ret (ex_intro Q x q)
31        | _ => raise TautoFail
32      end
33    end.

```

At a high level, the metaprogram `solve_tauto` finds the proof of proposition `P` in two steps: first, by looking for a hypothesis of type `P` in the list `l`, in which case that hypothesis constitutes the proof; and second, if it cannot find such an hypothesis, by decomposing the proposition `P` according to its shape, and recursively calling the solver on the resulting subcomponents of `P`.

We defer for the moment the explanation of the `lookup` metaprogram used to search the list of hypotheses (Line 3), and only point out that it happens before we inspect `P` to avoid unnecessarily taking apart propositions for which we already have a proof.

Mtac offers the `mmatch` construct to analyze the shape of arbitrary Rocq terms. It takes a scrutinee and a list of branches, each containing a unification candidate and a metaprogram. `mmatch` attempts to unify the scrutinee with each of these candidates. When a match is found, the code in the respective branch is executed. Note that subsequent failures inside the branch will *not* automatically cause Mtac to backtrack and try the

remaining branches. Mtac’s `mmatch` is thus closer to Ltac’s `lazymatch` and to OCaml’s pattern-matching than to Ltac’s `match`.

In Line 5, we `mmatch` the proposition `P` against 5 possible shapes, which represent the connectives our solver can handle: `True`, conjunction, disjunction, implication, and existential quantification. We also add a catch-all branch (Line 31) to explicitly raise a meaningful exception if the solver is given a connective which it does not support.

In each branch of `mmatch`, the `[? x1 .. xn]` syntax introduces meta-variables `x1 . . . xn` (also called unification variables or existential variables) which the unification algorithm can instantiate. These variables are available in the code of their respective branch.

The most interesting aspect of Mtac’s `mmatch` is that—unlike its Ltac counterpart `lazymatch`—it is *statically typed*. Moreover, like Rocq’s own `match`, Mtac’s `mmatch` is an instance of dependent pattern matching. First of all, the type of `mmatch` is expressed as a function of the scrutinee. Mtac imitates Rocq’s `match` syntax, allowing users to specify the type of `mmatch` with the familiar `mmatch x as x' return P x' with ..` syntax.⁴ Second, the type of each branch is specialized by substituting the scrutinee with the branch’s candidate.

As an illustration of the dependently-typed nature of Mtac’s `mmatch`, consider the case for `True` on Line 6: Instead of having to provide a proof of the abstract proposition `P`—a rather impossible task—we are allowed to construct a proof of `True` instead, since the typechecker is aware of the equivalence between `P` and `True`. Indeed, the `mmatch`’s return type, which we can think of as `fun P' => M P'`, has been instantiated with `True` to yield `M True`. This is exactly the type of `ret I`, where `I` is the constructor of `True`.

Not all cases are as simple as the one for `True`. We turn our attention towards the cases for conjunction and disjunction. To handle these, we want to recursively traverse `q1` and, if necessary, `q2`. To this end, Mtac offers a general fixed-point combinator, called `mfix`,⁵ which we can use to acquire proofs of `q1` and `q2`. In the case of a conjunction, combining these proofs into a proof of `P` amounts to only two bind operations and an application of the `conj` constructor. If any of the recursive calls raise an exception, it is automatically propagated outwards by the bind operator.

A different strategy is needed for disjunction: We do not know *a priori* which side of the disjunction we will be able to prove. This brings us to the first application of Mtac’s backtracking mechanism. To enable backtracking of the program state, Mtac offers the `mtry` construct, which executes a given program and handles any exceptions it raises according to a given list of handlers. These handlers behave similarly to `mmatch` branches.

The solver uses `mtry` in Line 12 to first attempt a proof of the left-hand side of the disjunction. If the recursive call is unable to prove that side of the disjunction, we simply try the other side.

The next case handled by our solver is implication. When proving an implication, we would like to add the left-hand side to a set of facts that can be used to solve the right-hand side. To this end, we use the list of assumptions `1`. This list contains assumptions of heterogeneous types. We

⁴ The `mmatch` syntax imposes a syntactic restriction on the return type: it has to be an application of `M`. This does not pose a limitation for any of the examples presented in the thesis.

⁵ For technical reasons, Mtac cannot provide just one fixed-point combinator for all arities and instead provides `mfix1` through `mfix5`.

⁶ In Rocq, an argument surrounded by curly braces is an *implicit argument* and, as such, should not be provided explicitly.

account for this heterogeneity using a universal type called `dyn` which is defined by a single constructor `Dyn`: `forall {T : Type}, T -> dyn`.⁶

It is worth pointing out that, in the case of implication, we will have to provide a proof term in the form of a function, i.e., `(fun q1 => ...)`. However, we need access to argument `q1` before we return such a function—otherwise, how could we possibly add `q1` to the list of assumptions for the recursive call? This seeming contradiction is resolved by Mtac’s `nu` construct:

```
nu: forall {X T : Type} (var : name) (x : option X),
    (X -> M T) -> M T.
```

`nu` introduces a new variable in the execution context of the program, having name `var` and type `x`. Then, it calls its continuation argument, passing it the newly-introduced variable. The result of calling `nu` is whatever the continuation returns. Optionally, the variable can be given a body, turning it into a *local definition*. The `name` type enables different ways of indicating the name of the variable. Its most basic constructor, `TheName`, simply contains a string. Mtac defines the notation `(ν x, t)` for `nu (TheName "x") (fun x => t)`.

⁷ This primitive was called `abs` in Mtac1 but has since been renamed to distinguish it from other abstraction primitives.

Working hand in hand with `nu` is Mtac’s `abs_fun`⁷ construct:

```
abs_fun: forall {X T}, X -> T -> M (X -> T).
```

The `abs_fun` construct attempts to convert a term `t`, which may mention variable `x`, into a function `fun x' => t[x'/x]`, where `t[x'/x]` represents the meta-theoretic substitution of `x` by `x'` in `t`.

A note on soundness.

The `nu` operator by itself appears to be unsound. It is easy to see that `ν (x : False), ret x` claims to produce a proof of `False` out of thin air. Fortunately, Mtac ensures soundness by dynamically checking that `ν` variables do not escape their scope [Zil+15].

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

To implement the implication case in our solver, we use `nu` to introduce a variable `q1 : Q1`, recursively call the solver with the list of assumptions extended by `q1`, and finally abstract away `q1` with `abs_fun`.

The one remaining connective is existential quantification. Our approach here is to introduce a meta-variable, recursively call the solver, and continue only if the meta-variable has been instantiated by the recursive call. In this way, we make sure to not leave any dangling meta-variables in our proof of `P`.

Mtac offers two primitives related to meta-variables. The first primitive is `ewar`, which is used to introduce meta-variables. Its type is:

```
ewar: forall {X : Type}, M X.
```

The second primitive is `is_ewar`, which checks if the given term is an *uninstantiated* meta-variable:

```
is_ewar: forall {X : Type}, X -> M bool.
```

These two primitives suffice to implement the case for existential quantification. However, we have not yet explained how the meta-variables we introduce are actually instantiated. Fortunately, we do not need to introduce additional code. The `lookup` function will instantiate these variables for us. To see why, consider the proposition given to the recursive call in the case of an existential quantification, $Q\ x$, where Q represents the body of the existential, as a function binding the witness, and x is the freshly introduced meta-variable. This recursive call, and any further recursive calls on sub-propositions of $Q\ x$, will call `lookup`, which in turn attempts to unify the proposition with each assumption. Crucially, this unification attempt—if successful—will instantiate the meta-variables in both sides of the unification task. Thus, by successfully matching up assumptions with the current proposition, `lookup` also takes care of instantiating meta-variables.

The `lookup` metaprogram is given below.

```

1 Definition NotFound : Exception. constructor. Qed.
2 Fixpoint lookup (P : Prop) (l : list dyn) : M P :=
3   match l with
4   | D :: l => mmatch D with [? (p : P)] Dyn p => ret p | _ => lookup P l end
5   | [] => raise NotFound
6   end.

```

We first define a new exception called `NotFound`. In `Mtac`, new exceptions are created with opaque definitions of the type `Exception`, whose definition is given below.

```
Inductive Exception := exception.
```

The `Exception` type is a proposition equivalent to `True`, *i.e.*, an inductive type with a single constructor that takes no arguments. To avoid relying on the name of that constructor, the `constructor` tactic is used, which takes care of selecting the constructor automatically.

To make each inhabitant of `Exception` distinguishable from the other inhabitants of this type, we conclude the definition of new exceptions with `Qed`. In `Rocq`, every definition concluded with `Qed` is opaque, *i.e.*, it cannot be unfolded by reduction. Hence, two distinct opaque identifiers of type `Exception` cannot be equated by the typechecker, even though their definitions are the same.

After creating the exception, we proceed to define the `lookup` function as a regular `Rocq` `fixpoint`, which iterates over the list of hypothesis (1). For each element D of l , the tactic matches D against any hypothesis of type P . We achieve this by specifying an `mmatch` pattern that introduces a new meta-variable for the proof of P called p . The pattern, with all implicit arguments given, is `[? (p : P)] @Dyn P p`. If the list element D matches this pattern, then D must be `@Dyn P p'` for some p' . The unification mechanism behind `mmatch` will instantiate p with p' and we return p . If no element in the list matches this pattern, we raise the `NotFound` exception.

Execution of `solve_tauto`:

Mtac includes a tactic `mrn` which executes its metaprogram argument on the current goal. Note the `_` to avoid writing the proposition: it is deduced from the goal.

```
Example test_tauto: 5 = 7 -> exists x, x = 7.  
Proof. mrn (solve_tauto [] _). Qed.
```

10

Mtac2: Adding backward reasoning to Mtac1

Now that we have seen the basics of Mtac, we are ready to present what is new in Mtac2, namely its support for *backward reasoning*. First, in this section, we introduce Mtac2’s support for Ltac-style *dynamically-typed* backward reasoning, and then continue in Section 11 by showing how Mtac2’s backward reasoning can be usefully combined with the *statically-typed* approach of Mtac1.

It is worth noting that supporting Ltac-style proofs in Mtac2 is not just a stepping stone towards statically-typed tactics. It is also useful in its own right because it provides Ltac-style tactic programming based on operational semantics more principled than those of Ltac, whose semantics are widely appreciated to be subpar within the Rocq community [Bar09; Péd19; Péd16b].

Additionally, another interesting point that should come across throughout this section is that we can build a realistic tactic language simply combining the meta-programming capabilities offered by Mtac with the several facilities from the Gallina language, like notations, dependent types and type classes. By working at a higher-level of abstraction than that supported by OCaml, our tactics are noticeably short and can be easily extended by the end user to support additional features.

In order to see how we support backward reasoning in Mtac2, let us start with a simple example—in Ltac—which is sufficient to highlight some of the important issues we had to resolve.

Example 18 (Tactics in Ltac).

```
Lemma sub_0_r : forall n, n - 0 = n.  
Proof.  
  intro n. case n; [ | intro n']; reflexivity.  
Qed.
```

The code above proves that subtracting 0 from any natural number n is equal to n . Since subtraction is defined by performing pattern matching on its first argument, this identity does not hold directly by computation. We must perform case analysis.

In this section we focus on how to build a similar proof script in Mtac2. For that, we need to take a sneak peek at how Rocq works internally. The

[Bar09] Barras, “The Dark Side of Ltac”

[Péd19] Pédro, “Ltac2: Tactical Warfare”

[Péd16b] Pédro, “Towards Ltac 2.0”

first thing to note about this code is that we have five *statements* (code followed by a period). The first one is stating the lemma, the second is the `Proof` command, which tells Rocq that we want to write a proof in Ltac, and the last one is `Qed`, which tells Rocq that we are done with the proof. Our interest is in what happens in between, in what we call the *tactic statements*. These statements transform the proof state to incrementally build a proof term.

After issuing the `Proof` command, Rocq produces a *goal*. Internally, a goal in Rocq is represented by a *meta-variable*, a hole in the proof term under construction. This meta-variable has a type, namely the lemma we have to prove. In this case, the meta-variable—let us call it $?g$ —created for the goal has type `forall n, n - 0 = n`.

For introducing the variable n , `intro` solves (instantiates) the meta-variable $?g$ with an abstraction `(fun n:nat => ?g1)`, where $?g_1$ is a new meta-variable of type `n - 0 = n`. But this type only makes sense in a local context where n is introduced. Following the tradition of Contextual Modal Type Theory [NPP08], we write such contexts as $?g_1 : (n:nat \vdash n - 0 = n)$, meaning that $?g_1$ is a meta-variable with type `n - 0 = n` in the local context `n:nat`.

Rocq executes each tactic statement under the local context of the current goal. For the first tactic statement, that was the empty context. For the second one, it is `n:nat`, the context of $?g_1$.

Continuing, the `case` tactic solves the goal ($?g_1$) with pattern matching (a `match`). The generated `match` contains two subgoals, one for each of its branches: $?g_{11} : (n:nat \vdash 0 - 0 = 0)$ and $?g_{12} : (n:nat \vdash \text{forall } n', (S\ n') - 0 = S\ n')$. The first one corresponds to the base case, in which n is `0`, and the second one corresponds to the inductive case, in which n is the successor S of some number n' .¹ The `case` tactic returns these two subgoals, which the composition operator (the semi-colon) composes with the tactics listed in `[ | intro n']`. The first subgoal is dealt with by the tactic to the left of the `|`, which is implicitly the identity tactic (`idtac`), while the second subgoal is dealt with by the tactic to the right of the `|`, `intro n'`. The output of this composition is, again, two subgoals: $?g_{11}$, unchanged, and the new subgoal $?g_{121} : (n:nat, n':nat \vdash (S\ n') - 0 = S\ n')$. Finally, the second semi-colon calls `reflexivity` on each of these subgoals, trivially solving them by computation.

In Mtac2 we can write the same proof as follows:

Example 19 (Proof of Example 18 written in Mtac2).

```
Lemma sub_0_r : forall n, n - 0 = n.
MProof.
  intro n. case n >> [m: idtac | intro n'] >> reflexivity.
Qed.
```

At a high level, the two proof scripts look similar. However, in contrast to the Ltac version, all the tactics used in the Mtac2 proof script are defined in Rocq. To understand how Mtac2 makes this possible, we describe the main building blocks of Mtac2:

1. The `tactic` type (Section 10.1).

[NPP08] Nanevski et al., “Contextual modal type theory”

¹ The numbering of subgoals is meant to reflect the derivation *tree* of the proof. Internally, Rocq manages lists of goals.

2. The `goal` type (Section 10.2).
3. A generalization of the `tactic` type (Section 10.4).
4. Tactic composition (Section 10.3).

To conclude this section, we will present several basic tactics constructed with the infrastructure developed throughout this section (Section 10.5).

10.1 THE `tactic` TYPE

Think of the `case` tactic as a procedure that takes the element to do case analysis on, and solves the current goal by creating subgoals for each constructor of the inductive type. If we want a functional interface, in which the current goal is explicitly given, the signature can be:

```
Definition case : forall {A}, A -> goal -> M (list goal) := ...
```

The first two arguments describe the (implicit) type `A` and the element of type `A` to do case analysis on. The third argument is the current goal, and the result is an `Mtac2` meta-program that returns a list of subgoals (or raises an exception if `A` is not a inductive type). Instead of explicitly mentioning goals, we can first define a type alias `tactic` and use that to give `case` a more intuitive type:

```
Definition tactic := goal -> M (list goal).
Definition case : forall {A}, A -> tactic := ...
```

Readers familiar with LCF-style provers will find this representation familiar. For the moment, we will consider tactics as having this type, but in Sections 10.2 and 10.4 we will enrich it to include different improvements, until we reach the actual type used in `Mtac2`. A different, but related notion of tactics—providing additional static guarantees—is presented in Section 11.

Even without the definition of the `goal` type, we can already present a few very basic tactics. The identity tactic takes a goal and returns it unmodified; the failing tactic takes an exception and throws it; and the `try` tactical executes its argument but silently ignores any errors it might raise.

```
Definition idtac : tactic := fun g => ret [g].
Definition fail (e: Exception) : tactic := fun g => raise e.
Definition try (t: tactic) : tactic := fun g =>
  mtry t g with _ => ret [g] end.
```

An `Mtac2` tactic statement can be executed using the provided `Ltac` tactic `mrtn`. For instance, in a `Proof` environment, calling `mrtn (try idtac)` will have the effect of executing the `Mtac2` tactic statement `try idtac` on the current goal, say `?g`. If and when the tactic (applied to the current goal) successfully finishes executing, it produces a list of goals (meta-variables), which the interpreter registers to `Rocq` as the new goals to solve next. In this case, the list will have the one goal `?g`. Alternatively, an `Mtac2` tactic statement can be executed using the `MProof` environment, which wraps any enclosed tactic into a `mrtn`.

10.2 THE goal TYPE

In Example 18, we saw an example goal and its evolution during the execution of an Ltac proof script. In this section, we describe how we represent goals and their evolution in Mtac2.

Figure 10.1: The goal type.

```

1 Inductive goal_state := gs_active | gs_any.
2
3 Inductive goal : goal_state -> Type :=
4   | Metavar (gs : goal_state) (A : Type) : A -> goal gs
5   | AHyp {A} : (A -> goal gs_any) -> goal gs_any
6   | HypRem {A} : A -> goal gs_any -> goal gs_any
7   | HypReplace {A B} : A -> (A = B) -> goal gs_any -> goal gs_any.

```

The main difficulty to overcome is that, as seen in Example 18, the goals (meta-variables) returned by a tactic generally live in different contexts, each of which contains its own changes—newly-introduced or deleted hypotheses—with respect to the initial context. To be able to return these goals as a list, we require a representation of their meta-variables that is valid in the initial context. The `goal` type presented in Figure 10.1 achieves exactly that. It does so by wrapping a goal’s meta-variable in constructors which represent the changes relative to the initial context, thus making the representation self-contained with respect to that context.

Additionally, the `goal` type statically informs us either that the goal at hand is in sync with the initial context—we say in this case that the goal is *active*—or that some work must be done to the context in order to adapt it to that of the goal. We achieve this by introducing a type refinement: the `goal` type is indexed by a new type `goal_state`, an enumeration with two values (Line 1 in Fig. 10.1). Any goal can be `gs_any`, but only `Metavar` can be `gs_active`, in which case no work needs to be done to use the goal in the initial context. With this in mind, we need to slightly modify the type of a tactic:

```

Definition tactic := goal gs_active -> M (list (goal gs_any)).

```

In essence, this type not only says that a tactic receives a goal, but that the goal is active (`gs_active`). As for the goals returned by the tactic, we cannot ensure they are active yet (`gs_any`). By establishing that the goal received by a tactic must be active we are freeing the tactic developer from the burden of syncing the contexts.

In the remainder of this section, we explain each constructor of the `goal` type in detail, and conclude by showing a function to activate a goal prior to passing it to a given tactic.

The first constructor, `Metavar`, is the base case of the inductive type, representing a goal that is well-formed in the *current* context.² There are two interesting points to note about `Metavar`. First, we of course cannot ensure statically that the term argument (of type `A`) is in fact a goal (i.e., an uninstantiated meta-variable) because Rocq’s type system does not have a way of expressing that a term is or is not a meta-variable. Rather, we make it an invariant of the Mtac2 implementation that tactics

² For presentation purposes, we have simplified this constructor a bit compared to the reality of its implementation in Rocq. The difference will be explained in detailed when we introduce *sort polymorphism* (Section 14.2).

only receive `Metavar`s whose arguments are uninstantiated meta-variables. Second, it might look suspicious that `Metavar` can be used to construct either a goal of type `goal gs_active` or `goal gs_any`. The reason for such freedom is rooted in the need to be able to refer to an active goal as one whose state is unknown, since this is what a tactic must return. What matters is that `Metavar` is the only constructor of active goals, and hence the only possible constructor for the argument to a tactic. As an example of the use of `Metavar`, in Example 19, the initial goal after `MProof` is `Metavar gs_active (forall n, n - 0 = n) ?g`.

The second constructor, `@AHyp P (fun H : P => fg)`, represents a single hypothesis `H` of type `P` introduced into the context of an inner goal `fg`, whose type may depend on `H`.³ (The `@` syntax allows us to specify the type of the binder `P` explicitly, even though it is marked as an implicit argument.) For example, after executing `intro n` in Example 19, the goal will be:

```
@AHyp nat (fun n : nat => Metavar gs_any (n - 0 = n) ?g1)
```

The third constructor, `HypRem x g`, indicates that hypothesis `x` is marked as removed. This mark makes it possible to implement tactics such as `clear`, which perform destructive updates on the proof context. Note that `HypRem` does not distinguish between hypotheses in the initial context and new hypotheses represented by `AHyp`. Consider, for the sake of illustration, that our initial goal is `Metavar gs_any (True -> n = n) ?g`, and that we introduce and immediately remove the hypothesis `H` with the proof script `cintros H {- clear H -}`. Here, for illustration purposes and for reasons that will become clear in the coming sections, we use the `cintros` tactic, which binds the hypothesis introduced `H` in the body surrounded by `{- -}`. Using `HypRem`, we represent the resulting goal with:

```
@AHyp True (fun H : True =>
  @HypRem True H (Metavar gs_any (n = n) ?g'))
)
```

Finally, the last constructor, `@HypReplace A B x eqAB g`, indicates that the type of hypothesis `x` (that is, `A`) changed to the equivalent type `B`. The equivalence is witnessed with the proof `eqAB`, which must be `eq_refl`—see Section 13.4.5. With this constructor, for instance, we are able to encode the effect of simplifying a hypothesis (`simpl in`, in the idiom of `Ltac`).

We have explained the intended meaning of the constructors, but we have not shown how we manipulate them to (1) solve goals, (2) create new goals, and (3) activate goals. We discuss these issues in the coming subsections.

10.2.1 Solving goals

Tactics assume they are given as input an *unsolved goal*, i.e., a goal whose metavariable is as yet uninstantiated. To *solve* this goal, tactics instantiate the meta-variable of the goal by unifying it with a term of the appropriate type.

As an example, the `exact` tactic listed in Figure 10.2 takes a term `x` of some type `A` and instantiates (unifies) the goal's meta-variable `e` with `x`.

³ An additional constructor—omitted for brevity—handles let-bindings in the goal, as well as local definitions such as those introduced by the `pose` tactic.

Figure 10.2: `exact` tactic to solve a goal with a given term.

```

1 Definition exact {A} (x:A) : tactic := fun g =>
2   match g with
3   | Metavar _ _ e => cumul_or_fail UniCoq x e;; ret []
4   end.

```

Note that, thanks to how the type `tactic` is defined, the goal `g` has type `goal gs_active`, making `Metavar` the only possible constructor for it. The call to `cumul_or_fail` (Line 3) then unifies `x` with `e`. `cumul_or_fail` is just a wrapper over `cumul`, which takes two terms of potentially distinct types and unifies them after *cumulatively* unifying their types first (using the Mtac2 primitive `unify_univ` to that effect).⁴ `exact` returns an empty list of goals: if no error is produced by `cumul_or_fail`, we know `e` is instantiated with `x` and no goal is left.

⁴ *Cumulativity* is related to type universes: instead of restricting the type of `e` to be exactly the one of `x`, we allow for it being *larger* in the hierarchy of types.

10.2.2 Creating new goals

We have already seen that Mtac2 tactics transform a given input goal into (potentially) multiple subgoals. To create a new subgoal, tactic developers use the `ewar` primitive to generate a fresh meta-variable, wrap this meta-variable in the `Metavar` constructor, and, if the context changes, wrap the result in another of the constructors of the `goal` type. The resulting goal is then returned as part of the list of goals produced by the tactic.

Figure 10.3: `split` tactic creating two subgoals.

```

1 Definition split : tactic := fun g=>
2   P <- goal_prop g;
3   mmatch P with
4   | [? Q R] Q /\ R =>
5     eQ <- ewar Q;
6     eR <- ewar R;
7     exact (conj eQ eR) g;;
8     ret [Metavar gs_any _ eQ; Metavar gs_any _ eR]
9   | _ => raise NotAConjunction
10  end.

```

Consider a simplified version of the `split` tactic, listed in Figure 10.3. Given a proposition of the form $Q \wedge R$, it creates meta-variables `eQ` and `eR` for `Q` and `R`, and instantiates the current goal with the `conj` constructor of the $\wedge$ operator applied to `eQ` and `eR`. Then, it returns the list containing the two meta-variables wrapped in the goal constructor `Metavar`.

10.2.3 Activating a goal

If a tactic takes as input an active goal but returns goals in potentially different contexts, how is it possible to apply another tactic to any of the goals returned? The answer is in the function `activate_and_apply` (Figure 10.4), which ensures that a given tactic `t` is called with the `Metavar` constructor. We call this process *activating* the goal. The function `activate_and_apply` activates the goal `g` by recursively traversing `g`, and, at every step, updating the proof context in accordance with the outermost constructor of `g`. For `AHyp`, Mtac's `nu` primitive is used to extend the context

```

1 Definition activate_and_apply (t: tactic A) :
2   goal gs_any -> M (list (goal gs_any)) :=
3   fix activate g :=
4     match g return M _ with
5     | Metavar _ _ e => t (Metavar gs_active _ e)
6     | @AHyp C f =>
7       nu (FreshFrom f) None (fun x : C =>
8         activate (f x) >>= add_hyp x
9       )
10    | HypRem x f => remove x (activate f) >>= add_rem x
11    | HypReplace x e f => replace x e (activate f) >>= add_rep x e
12  end.

```

Figure 10.4: Implementation of `activate_and_apply`, a function to apply a tactic to a goal.

with an additional hypothesis (Line 7), whose name is chosen to match the binder of the inner goal `f` using the `FreshFrom` constructor of the name type. For the remaining cases, we make use of two `Mtac` primitives that we have not yet presented but which are described in detail in Section 13. In the case of `HypRem`, we use `remove` to delete the hypothesis from the context (Line 10). Here, `remove` implements strengthening: the continuation is executed in a context without the variable `x`. Similarly, in the case of `HypReplace`, we use the primitive `replace` to replace the type of a hypothesis with an equivalent one (Line 11). In both of these cases, the primitive makes well-bracketed changes to the proof state: it modifies the proof context, executes the function it is passed under the modified context, and then undoes the modification of the proof context before returning.

When `activate_and_apply` reaches the base case, `Metavar`, it executes the tactic `t` with the goal’s meta-variable wrapped in the same constructor, but with `gs_active` instead of `gs_any`. The execution of `t` now takes place in the modified proof context that is in sync with that of the given goal `g`.

After `t` is done executing, we *replay* all the constructors of the initial goal `g` in the goals resulting from the execution of `t`. That is, we replicate every occurrence of `AHyp`, `HypRem`, and `HypReplace` from the initial `g` in every goal. This is done by calls to `add_hyp` (Line 8), `add_rem` (Line 10), and `add_rep` (Line 11), respectively. The code for these functions is given below. They simply wrap the respective constructor around each goal in the list of goals they are given.

```

1 Definition add_hyp {B} (y : B)
2   : list (goal gs_any) -> M (list (goal gs_any)) :=
3   M.map (fun g => r <- abs_fun y g; ret (@AHyp B r)).
4 Definition add_rem {B} (x : B) (l: list (goal gs_any))
5   : M (list (goal gs_any)) :=
6   ret (List.map (HypRem x) l).
7 Definition add_rep {A B} (x : A) (e : A = B) (l: list (goal gs_any))
8   : M (list (goal gs_any)) :=
9   ret (List.map (HypReplace x e) l).

```

In the definition of `add_hyp`, the reason that `Mtac`’s `M.map` function is used instead of the normal Gallina `List.map` is that the function being mapped over the goals is not a pure Gallina function but an `Mtac` metaprogram (it uses `abs_fun` to abstract the goals over the hypothesis `y`).

10.3 TACTIC COMPOSITION

In Mtac2, a tactic can be composed with different objects, obtaining a new tactic, which—in turn—can be composed again. We can compose a tactic with:

A tactic. For instance, consider the composition `case n &> reflexivity`. Here, `reflexivity` is applied to every subgoal introduced by the case analysis.

A list of tactics. For instance, in Example 19, we compose `case n` with the list⁵ of tactics `[m: idtac | intro n']`. Here, each of the tactics in the list is applied to the corresponding subgoal introduced by the case analysis.

A goal selector. A *goal selector* is a tactic combinator that selects a—potentially reordered—subset of unsolved goals to be given to the following tactic. For instance, the following code features `first`, a goal selector that applies a given tactic only to the first subgoal, leaving the rest unchanged: `case n &> first reflexivity`.

We show in detail each case, and show how to overload the `&>` operator to represent them all.

Composing two tactics.

For the simple case of composing two tactics, the syntax is implemented by the `tactic_tactic` function shown in Figure 10.5, along with two helper functions (`is_unsolved` and `filter_solved_goals`) which ensure that `t2` is never called on a solved goal (*i.e.*, on an instantiated meta-variable). This is necessary because meta-variable instantiation can happen as a side effect of *any* unification operation. Thus, the point in time right before executing a tactic on a goal is the only point at which we can meaningfully check if the corresponding meta-variable has already been instantiated, in which case we do not need to run the tactic.

Figure 10.5: The implementation of the tactic composition operation `tactic_tactic` and its helper functions `is_unsolved` and `filter_solved_goals`.

```

1 Definition is_unsolved : forall {gs}, goal gs -> M bool :=
2   mfix2 unsolved (gs : goal_state) (g : goal gs) : M bool :=
3     match g with
4     | Metavar _ _ e => is_evar e
5     | AHyp f => nu Generate None (fun x => is_unsolved _ (f x))
6     | HypRem _ g => is_unsolved _ g
7     | HypReplace _ _ g => is_unsolved _ g
8     end.
9
10 Definition filter_solved_goals := M.filter (@is_unsolved gs_active).
11
12 Definition tactic_tactic (t1 t2 : tactic) : tactic := fun g =>
13   gs_out1 <- t1 g;
14   gs_in   <- filter_solved_goals gs_out1;
15   gss_out2 <- M.map (activate_and_apply t2) gs_in;
16   ret (concat gss_out2).
```

The first step of `tactic_tactic` is to execute `t1` on the given goal `g`. The resulting goals `gs_out1` are then filtered to remove solved goals with the help of `filter_solved_goals`,⁶ yielding `gs_in`. At this point, the goals are of type `goal gs_any` and cannot simply be given to `t2`. Thus, instead of mapping `t2` over `gs_in`, we make use of `activate_and_apply` to activate each single goal in `gs_in` before passing it to `t2`. We are left with a list of lists of goals which we flatten (using Rocq's `concat` function) to arrive at the final list of goals to return.

⁶`filter_solved_goals` makes use of a monadic filter function on lists, which we write `M.filter` to disambiguate it from Rocq's own filter function on lists.

Composing a tactic with a list of tactics.

In Example 19, we have seen tactics being composed with a *list of tactics*: `case n &> [m: idtac | intro n']`. This composition operator, in general, applies the i^{th} follow-up tactic to the i^{th} subgoal returned by the initial tactic. In `Mtac2`, this form of composition is implemented by the `tactic_list` function shown in Figure 10.6 and its helper function `tmap`.

```

1 Fixpoint tmap (tacs : list tactic) (gs : list (goal gs_any)) :
2   M (list (list (goal gs_any))) :=
3   match tacs, gs with
4   | [], [] => ret []
5   | tac :: tacs', g :: gs' =>
6     gs_out <- activate_and_apply tac g;
7     gs_rec <- tmap tacs' gs';
8     ret (gs_out :: gs_rec)
9   | l, l' => raise NotSameSize
10  end.
11
12 Definition tactic_list (t : tactic) (ts : list tactic) : tactic := fun g =>
13   gs_out1 <- t g;
14   gs_in <- filter_solved_goals gs_out1;
15   gss_out2 <- tmap ts gs_in;
16   ret (concat gss_out2).

```

Figure 10.6: Composing a tactic with a list of tactics.

`tactic_list` is similar in structure to `tactic_one`. The only difference is that the list goals and the list follow-up tactics are zipped (implicitly) by `tmap`. Every tactic is applied to its respective goal using the `activate_and_apply` function. Should the lengths of the lists not agree, `tmap` raises an exception (`NotSameSize`).

Composing a tactic with a goal selector.

With `tactic_list` allowing us to assign follow-up tactics to *all* goals, it is natural to ask for a way to focus on *one* goal or a *subset* of goals. This functionality is provided by what we call *goal selectors*. We define a Rocq type of goal selectors called `selector`:

```
Definition selector := list (goal gs_any) -> M (list (goal gs_any)).
```

With this type in mind, we can easily define selectors such as `nth n`, which executes a given tactic on the n^{th} goal, and `rev`, which reverses the list of goals. The code for both selectors is given below. The function `split_at n 1` used on Line 2 splits a list `l` in two, the first one having at

most the first n elements, and the second having the remainder of the list. Note that the new subgoals produced by $f\ x$ are placed where the original goal was in the list l .

```

1 Definition nth (n : nat) (t : tactic) : selector := fun gs =>
2   let (l1, l2) := split_at n l in
3   match hd_error l2 with
4   | None => raise NoGoalsLeft
5   | Some (x, g) =>
6     goals <- activate_and_apply (f x) g;
7     filter_solved_goals (l1 ++ goals ++ tl l2)
8   end.
9
10 Definition rev : selector := fun l => ret (List.rev l).

```

Finally, we define how selectors are to be composed with tactics. This composition is implemented by `tactic_selector`, whose code is given below. The implementation executes the tactic t , filters out solved goals, and then passes the remaining goals to the selector s .

```

Definition tactic_selector (t : tactic) (sel : selector) : tactic := fun g =>
  gs_out <- t g;
  gs_in <- filter_solved_goals gs_out;
  sel gs_in.

```

This composition operation, with the appropriate overloading for the `&>` operator, allows us to write tactics such as `case n &> first reflexivity`. Here, `first` is simply a shorthand for `nth 1`.

Overloading composition.

As we have seen above, `Mtac2`'s composition operator, `&>`, is overloaded. We use type classes [SO08] to give different meaning to the operator depending on type of the second operand. The type class behind this is called `Seq` and its definition is given below. It is indexed by the type of the right-hand side operand (C). Its only field is `seq`, which represents the composition operation. The composition operator `&>` is simply a shorthand for `seq`.

```

Class Seq C := seq : tactic -> C -> tactic.

```

The three composition variants presented above can be written as instances of this type class:

```

Instance seq_tactic : Seq tactic := { | seq := tactic_tactic | }.
Instance seq_list : Seq (list tactic) := { | seq := tactic_list | }.
Instance seq_selector : Seq selector := { | seq := tactic_selector | }.

```

While this type class-based approach offers considerable flexibility for overloading the composition operator, it is somewhat limited in the kind of composition it allows: the list of goals is the only output that flows from the left-hand side to right-hand side of the composition operator. In the next subsection, we show how to generalize tactic composition to also allow *values* to be passed from one tactic to another.

10.4 RETURNING VALUES FROM TACTICS

The `tactic` type as it is defined now does not allow for values to be passed from one tactic to another: tactics only return goals. While continuation-passing style can be used to work around this issue (as is necessary in Ltac, see Chapter 14.3 [Chl11]), we can offer a more ergonomic solution by introducing a generalized type for tactics called `gtactic`. This more general type includes a return value. A type equivalent to `tactic` can thus be derived by returning values of type `unit`. Both definitions are given below.

```
Definition gtactic (A: Type) :=
  goal gs_active -> M (list (A * goal gs_any)).
Definition tactic := gtactic unit.
```

[Chl11] Chlipala, “Certified Programming with Dependent Types”

Note that a `gtactic` returns one value *for each resulting goal*, instead of one single value. As a result, the type `gtactic` forms a monad (as in [Spi10]—an exception monad with state for the meta-variables). The unit element is the identity tactic `idtac`—generalized appropriately to take an element of any type and return it together with the unchanged goal. To keep in line with the monadic perspective we call this function `ret`. Its implementation is shown below, where we use `M.ret` to mean Mtac’s primitive.

[Spi10] Spiwack, “An abstract type for constructing tactics in Coq”

```
Definition ret {A} (x : A) : gtactic A := fun '(Metavar _ _ g) =>
  M.ret [(x, Metavar gs_any _ g)].
```

The monad’s bind operation corresponds very closely to the composition of two tactics (`tactic_tactic`) we have seen before. We give its implementation in Figure 10.7. The only differences with `tactic_tactic` being the handling of the values (of type `A`) returned by the left-hand side tactic `t`. These values are passed to the right-hand side tactic `f`, which is now a function from `A` to `gtactic B`. In the typical scenario, `t` returns only one pair value-goal.

```
1 Definition bind {A B} (t: gtactic A) (f: A -> gtactic B) :
2   gtactic B := fun g =>
3   gs_out1 <- t g;
4   gs_in   <- filter_solved_goals gs_out1;
5   gs_out2 <- M.map (fun '(x, g') =>
6     activate_and_apply (f x) g'
7   ) gs_in;
8   ret (concat gs_out2).
```

Figure 10.7: The bind operator for tactics.

The ability to return values from tactics represents the final building block of Mtac2’s tactic infrastructure. Most definitions presented in terms of `tactic` can be generalized to `gtactic`. For brevity’s sake we do not restate them. In the following examples of tactics we prefer to use `tactic` instead of `gtactic` to avoid syntactic noise from extra parameters. Nonetheless, many of these examples can also be generalized to `gtactic`.

10.5 EXAMPLES

In this section we present several examples of basic tactics highlighting the different components of the tactic machinery of Mtac2 presented so far.

10.5.1 Introducing a hypothesis: the `cintro` tactic

Figure 10.8: The `cintro` tactic.

```

1 Definition cintro {A} (var: name) (f: A -> tactic) :
2   tactic := fun g =>
3     mmatch g with
4     | [?(P:A->Type) e] Metavar gs_active (forall x:A => P x) e =>
5       nu var None (fun x=>
6         let Px := reduce (RedWhd [RedBeta]) (P x) in
7         e' <- evar Px;
8         nG <- abs_fun (P:=P) x e';
9         _ <- exact nG g;
10        f x (Metavar gs_active _ e') >>= add_hyp x)
11     | _ => raise NotAProduct
12 end.

```

The `cintro` tactic shown in Figure 10.8 is an excerpt of the basic tactic for introduction of a hypothesis in Mtac2. We used it already in Example 19, although in its sugared form: `intro n` calls `cintro` with the name of `n` and `(fun _ => idtac)`.

This tactic takes the name `var` of the variable being introduced, and a continuation `f`. It proceeds by matching the goal against a `Metavar` constructor whose type is a dependent product indexed by type `A`. The body of the product is captured with the type predicate `P`: if the type is `forall x:A, b`, then `P` will be `fun x:A=>b`.⁷ If the unification succeeds, we introduce the variable into the context with the `nu` operator. With the new variable `x` in context, we create a new meta-variable `e'` with type `P x` after β -reducing it—we do not want our goals β -expanded. (The reduction mechanism of Mtac2 is explained in Section 13.4.2.) Then, variable `x` is abstracted from `e'`, effectively creating a function whose body is `e'`. We solve the current goal `g` with this term using the `exact` tactic. Having solved the current goal, we proceed to call the continuation tactic `f` with `x` and the new goal `Metavar gs_active _ e'`. Before returning, we abstract over `x` and add the corresponding `AHyp` constructor to all goals generated by `f x`. This is achieved using the `add_hyp` function presented in Section 10.2.3.

Note how, thanks to the invariant upheld by the `goal` type, the goal `g` must be a `Metavar`. There are no other cases, so if the `mmatch` fails, it is because the goal is not a dependent product, in which case we raise the exception `NotAProduct` (Line 11).

10.5.2 Goal pattern-matching

An important feature in tactic languages is the ability to inspect the proof goal. Here, we show how Mtac2 lets us *program* support for this feature

⁷The reason why unification will produce the most general instantiation for `P` is that `P x` is in the *higher-order pattern fragment*. Details can be found in [ZS17].

in the form of a tactic we call `match_goal`. At the moment, types will just help us avoid writing nonsensical patterns, but in Section 11 we flow into our typed tactics the typing information obtained by matching on the goal.

To begin with, consider the following example illustrating the use of `match_goal`. The example is deliberately simple and contrived, in order to quickly illustrate some of the features of goal pattern-matching.

Example 20 (Goal pattern-matching).

```

1 Example ex_goal_patt (x y z: nat) : x > 0 -> x + y > 0 ∨ z > 0.
2 MProof.
3   intros.
4   match_goal with
5   | [[? W L | w |- L ∨ w > W]] => left &> clear w
6   end.
7   (* proof continues *)

```

We need to prove that either $x + y > 0$ or $z > 0$ under the assumption that $x > 0$. In this case, after introducing x , y , and z into the proof context, we use `match_goal` to pattern-match on the goal and focus on the actual part of the proof relevant to our problem ($x + y > 0$), removing the remaining bits from the goal ($z > 0$). In the specified pattern, we indicate that the goal should be of the form $L \vee w > W$, where W and L are arbitrary, and w is a hypothesis (of type `nat`)—if this is the case, the pattern match succeeds.

More generally, the identifiers to the left of the `|` represent fresh meta-variables, the identifiers in between `|` and `|-` represent hypotheses, and the term to the right of `|-` represents the goal pattern. Correspondingly, the implementation of `match_goal` will deploy the following steps in implementing a pattern match:

1. Create a fresh meta-variable for every meta-variable listed to the left of `|`.
2. For every hypothesis listed between `|` and `|-`, search for a corresponding hypothesis that matches its type (prioritizing variables introduced earlier). In the case of w above, we know it must have type `nat`, because of its occurrence in $w > W$. If the type cannot be determined, Rocq will create a meta-variable for its type, and the algorithm will just match the first available hypothesis.
3. Unify the goal pattern (to the right of `|-`) with the actual goal. Upon success, execute the tactic corresponding to that branch of the pattern match. Else, backtrack.

Applied to Example 20, we get the following steps:

- 1 Create meta-variables `?W` (of type `nat`) and `?L` (of an unspecified type).
- 2 Search for a hypothesis of type `nat` and bind it to w . Pick x .
- 3 Unify the pattern `?L ∨ x > ?W` with the goal. Fail and backtrack.
- 2a Pick y .

3a Unify $?L \setminus y > ?W$ with the goal. Fail and backtrack.

2b Pick z .

3b Unify $?L \setminus z > ?W$ with the goal. Succeed and execute `left &> clear z`.

Figure 10.9: Goal pattern-matching.

```

1 Inductive goal_pattern : Prop :=
2   | gbase : forall {A}, A -> tactic -> goal_pattern
3   | gtele : forall {C}, (C -> goal_pattern) -> goal_pattern
4   | gevar : forall {C}, (C -> goal_pattern) -> goal_pattern.
5
6 Fixpoint match_goal_pattern'
7   (p : goal_pattern) : list Hyp -> list Hyp -> tactic :=
8   fix go l1 l2 g :=
9     match p, l2 with
10    | gbase P t, _ =>
11      gT <- goal_type g;
12      mif cumul UniCoq P gT then t g
13      else raise DoesNotMatchGoal
14    | @gtele _ C f, @ahyp A a d :: l2' =>
15      oeqCA <- unify C A UniCoq;
16      match oeqCA with
17      | Some eqCA =>
18        let a' :=
19          reduce RedNF (match eq_sym eqCA with eq_refl => a end)
20        in
21          mtry match_goal_pattern' u (f a') [] (rev_append l1 l2') g
22          with DoesNotMatchGoal =>
23            go (ahyp a d :: l1) l2' g
24          end
25      | None => go (ahyp a d :: l1) l2' g
26      end
27    | @gevar _ C f, _ =>
28      e <- evar C;
29      match_goal_pattern' u (f e) l1 l2 g
30    | _, _ => raise DoesNotMatchGoal
31  end.
32
33 Definition match_goal_pattern
34   (p : goal_pattern) : tactic := fun g =>
35   r <- hypos; match_goal_pattern' p [] (rev r) g.

```

`match_goal` is just notation for `match_goal_pattern` (Figure 10.9, Line 33); it takes a goal pattern p and in turn calls the auxiliary function `match_goal_pattern'`. This auxiliary function additionally takes as input two lists of hypotheses: $l1$, an accumulated list of already-visited ones, and $l2$, a list of yet-to-be-visited ones (in reverse order, *i.e.*, earlier-introduced hypotheses first, so that we process hypotheses from old to new). These two lists of hypotheses effectively represent a “cursor”, which we use during Step 2 of the pattern match in order to keep track of which actual hypotheses we have already tried to match against the hypothesis pattern currently being examined. (Thus, initially, the first list is empty, and the second list is the full list of hypotheses—*hypos*, see Section 13—in reversed (old-to-new) order.)

`match_goal_pattern'` is a double fixed-point: the outer one recurses over the pattern, and the inner one recurses over the list `l2`.

A goal pattern (Line 1) has three constructors, one for each of the steps. Step 1, the creation of a meta-variable, is represented by the constructor `gevar`, which takes the type `c` of the meta-variable and a continuation `f`. In Line 27 we can see how the algorithm `match_goal_pattern'` proceeds in this case: it creates a meta-variable `e` and calls itself recursively with `f` applied to `e`.

Step 2, the inspection of hypotheses, is represented by a similar constructor, `gtele`, also taking a type `c` and a continuation `f`. This case is a bit more complex, since we need to backtrack to try different hypotheses (Line 14). First, the type `c` is unified with `A`, the type of the first hypothesis in `l2`. If it fails (Line 25), it recurses, putting the hypothesis in the accumulator `l1` (i.e., moving the cursor to the next hypothesis). Else, it tries recursing with the pattern obtained from applying `f` to `a'`, a version of `a` with its type converted to `c`. In this case, we remove `a` from the list of hypotheses and move the cursor back to the beginning of the hypothesis list, since all remaining hypotheses should be considered for any subsequent hypothesis pattern matches. If in the recursive call an exception is raised, we backtrack, moving the hypothesis `a` into the accumulator instead (Line 22).

Finally, Step 3, the unification of the goal, is represented by the base constructor `gbase`, which takes the goal pattern `P` and a tactic `t`. The algorithm unifies `P` with the type `gT` of the goal `g` (`goal_type` just extracts the type from the `Metavar` constructor of the goal). If unification succeeds, we execute the tactic `t` with the goal `g` (Line 12); else we raise an exception in order to backtrack (Line 13).

The pattern from Example 20 is *desugared* to:

```

1  gevar (fun W=>
2    gevar (fun L=>
3      gtele (fun w=>
4        gbase (L \ / w > W) (left &> clear w))))

```

The select and assumption tactics.

Goal pattern matching allows for a very concise description of two basic tactics provided by `Mtac2`: `select`, which searches a hypotheses of a given type, and `assumption`, which searches for a hypothesis that will solve the current goal.

The tactic `select T` returns the first (oldest) hypothesis that matches the type `T`:

```

Definition select (T : Type) : gtactic T :=
  A <- goal_type;
  match_goal with [[ x : T |- A ]] => T.ret x end.

```

The tactic `assumption` solves the goal with a hypothesis of the same type:

```

Definition assumption : tactic :=
  A <- goal_type;

```

```
match_goal with [[ x : A |- A ]] => exact x end.
```

10.6 CONCLUDING REMARKS

Figure 10.10: A simple tautology solver in Mtac2 using backward reasoning.

```
1 Definition solve_tauto_mtac2 : tactic :=
2   mfix0 solve_tauto :=
3     assumption || exact I || (split &> solve_tauto)
4     || (left &> solve_tauto) || (right &> solve_tauto)
5     || (introsn_cont solve_tauto 1)
6     || (eexists |1> solve_tauto) || raise TautoFail.
```

With the type for tactics presented in this section, we are now able to implement our tautology solver in a style closer to what an Ltac user would be familiar with, using backward-reasoning tactics (Figure 10.10). Without going deeply into all the details, let us compare the case for conjunction. In the original Mtac1 code from Figure 9.1 we obtain a proof for each proposition in the conjunction, and return the `conj` constructor applied to those proofs. Instead, in Figure 10.10 we call the tactic `split`, which in essence applies the `conj` constructor, generating two subgoals, one for each of the propositions. Then, thanks to the composition operator `&>`, we solve each subgoal by recursively applying the tactic.

At this point, the reader may rightfully wonder what has been gained over just writing this tautology solver in Ltac. Indeed, there is not much difference between the Mtac2 code shown in Figure 10.10 and what one would naturally code up in Ltac. However, as we mentioned in the opening of this section, there are two major benefits of working with Mtac2. The first one is in semantics: our tactics, by virtue of being typed and being executed soundly—see Section 13—report errors at an earlier stage, making faulty tactics easier to debug.

For instance, an Ltac tactic like `clear x; (some_tactic_code; exact x)` will fail in the last tactic invocation (`exact`) with the error that `x` is not in scope. Instead, the Mtac2 variant (replacing `;` with `&>`) will fail when executing the first `&>`, prior to executing the code between parentheses.

The second benefit of using the tactic infrastructure of Mtac2 is extensibility: the end user might extend tactics without having to resort to OCaml. Take for instance the `&>` operator: we saw how to extend it to support goal selectors, without requiring any extra support from OCaml.

This said, in the next section we will go one step further and see how we can use the infrastructure built up thus far in order to develop a type of tactics—and another version of the tautology solver—that fruitfully synthesize the backward-reasoning style of Ltac with the much stronger static guarantees about tactic correctness afforded by Mtac1.

11

Typed backward reasoning

```
1 Definition solve_tauto : forall (P:Prop), ttac P :=
2   mfix1 solve_tauto (P : Prop) : M _ :=
3   mmatch P in Prop as P' return M (P' * _) with
4   | True => TT.apply I
5   | [? Q1 Q2] Q1 ∧ Q2 =>
6     TT.apply (@conj _ _) <*> solve_tauto Q1 <*> solve_tauto Q2
7   | [? Q1 Q2] Q1 ∨ Q2 =>
8     mtry
9       q1 <- TT.apply (@or_introl _ _) <*> solve_tauto Q1;
10      mif has_open_subgoals q1 then raise TautoFail else ret q1
11    with TautoFail =>
12      mtry
13        q2 <- TT.apply (@or_intror _ _) <*> solve_tauto Q2;
14        mif has_open_subgoals q2 then raise OpenGoals else ret q2
15      with OpenGoals => TT.demote end
16    end
17 | [? (Q1 Q2 : Prop)] Q1 -> Q2 =>
18   tintro (fun x:Q1 => solve_tauto Q2)
19 | [? X (Q : X -> Prop)] (exists x : X, Q x) =>
20   x <- evar X;
21   TT.apply (@ex_intro _ _ _) <*> solve_tauto (Q x)
22 | _ => TT.try (TT.use T.assumption)
23 end.
```

Figure 11.1: A simple tautology solver in Mtac2 using typed tactics.

The type for tactics provided in the previous section is not sufficiently expressive: a *tactic* conveys no information in its type, neither about the goals it solves nor the subgoals it creates. And some tactics may indeed not have anything to say about that. But there is no need to throw out the baby with the bath water: in this section, we show how Mtac2 also supports the construction of strongly *typed tactics*.

Typed tactics carry *static* information about the goals they solve and the subgoals they create in their type. Additionally, they are allowed to create *dynamic* subgoals whose type is only known at execution time. Moreover, with the infrastructure we have introduced so far, the implementation of typed tactics is surprisingly straightforward. We focus in this section on the key building blocks of typed tactics, and how to use them to build a strongly-typed version of the backward-reasoning tautology solver. We then present a more “real-world” example of typed tactics in Section 12.

Let us look at the main features of typed tactics in action. Figure 11.1

contains an implementation of the tautology solver using typed tactics. While the implementation may appear a bit verbose compared to its dynamically-typed counterpart in Figure 10.10, we will soon see how its strong typing makes it significantly more robust.

The type of typed tactics.

The most important feature of typed tactics is that their type mentions the goal they are applicable to. This is achieved by parameterizing the type of typed tactics, called `ttac`, by the corresponding type of goal:

Definition `ttac A := M (A * list (goal gs_any))`.

We call the parameter `A` the *static goal*, to distinguish it from the `goal` type introduced in the previous section. Regular goals can be thought of as *dynamic goals*, as they need to be dynamically inspected to find the type of the contained goal. In Figure 11.1, `solve_tauto`'s static goal is the universally quantified proposition `P`.

A typed tactic produces two components: (i) a proof of `A`, and (ii) a list of dynamic goals. The first component mirrors the way that `Mtac1` meta-programs generate proofs, whereas the second component bridges the gap between these meta-programs and `Mtac2` tactics. We explain how dynamic goals are useful in typed tactics further below.

The parameter `A` serves two purposes: (i) documenting what the typed tactic does, and (ii) guiding the implementation of the typed tactic itself. It is the second point that brings us to the next, and arguably most important, feature of typed tactics.

Statically-checked application of lemmas.

Even basic tactics such as `split`, `left`, `right`, and `exist` are essentially wrappers around constructors of inductive types or—put differently—lemmas that express the introduction rules for these types. These lemmas have informative types, but wrapping them in dynamically-typed tactics hides these types. This issue becomes even more apparent when we think about the `exact` and `apply` tactics, whose invocation documents very explicitly the lemma used and, therefore, the type of goals they can be applied to. Typed tactics aim to exploit this statically-available type information to provide a very strong guarantee: If Rocq's typechecker accepts the typed tactic, every application of a lemma contained within matches the type of the respective static goal.

We will now explain how `Mtac2` achieves these guarantees for typed tactics using the tautology solver of Figure 11.1. Note that, in the following, we define several typed tactics that have the same name as existing dynamically-typed tactics. To distinguish them, we prefix dynamically-typed tactics with the `T` namespace (for Tactics) and statically-typed tactics with the `TT` namespace (for Typed Tactics).

Let us first consider the case in which a lemma exactly solves the static goal at hand. For this case, `Mtac2` offers the `TT.apply` function whose code is given below. It returns the given lemma (as the proof of the static

goal) and no additional dynamic goals, as indicated by the empty list `[m:]`.¹

Definition `apply {A} (a : A) : ttac A := ret (a, [m:])`.

An example of `TT.apply` can be seen in Line 4, where the constructor of `True`, called `I`, solves the static goal immediately. It bears repeating that attempting to `TT.apply` a lemma whose type is not `True` would result in a static typechecking error at definition time of the tactic.

We consider the case presented above the base case of applying lemmas and build on that to handle the more general case of a `ttac A` applying a lemma of type `X -> .. -> Y -> A`. To this end, we introduce the *typed specialization operator* `tspec`. The operator is left-associative and is written `<*>`.

Consider the conjunction case of our typed-tactic version of `solve_tauto`. The constructor `conj` expects proofs of `Q1` and `Q2`. In Line 6, we use `tspec` to delegate these two hypotheses to recursive calls to `solve_tauto`, much in the same way that, in a hypothetical Ltac implementation of the solver, one could write `apply (@conj _ _); [solve_tauto|solve_tauto]`.

However, typed tactics give us static guarantees that dynamically-typed Ltac-style backward reasoning cannot provide. The type of the static goal in this branch of the solver is `Q1 ∧ Q2`. This matches the conclusion of `@conj _ _` (where Rocq infers the underscores to be `Q1` and `Q2`), whose type is `Q1 -> Q2 -> Q1 ∧ Q2`. The first application of `tspec` with `solve_tauto Q1` refines this to `Q2 -> Q1 ∧ Q2`. The second application of `tspec` provides the one remaining argument of type `Q2`. The remaining type, `Q1 ∧ Q2`, matches the static goal expected in this branch. (Note that we can elide the arguments to recursive calls: they can be inferred.)

If we were to pass a proof of `Q2` instead of `Q1` to the `TT.apply` invocation, the typechecker would reject our definition. If we were to omit either argument, the typechecker would reject our definition. If we were to apply `tspec` a third time, the typechecker would reject our definition. In short, the typed tactic combinators `apply` and `tspec` afford us the same kinds of static guarantees that we expect from function application in strongly-typed functional languages.²

The implementation of `tspec` is given below. `tspec` executes its arguments, typed tactics `f` and `x`, from left to right and collects their dynamic subgoals. As the proof term generated by the typed tactic `f` is of type `A -> B`, we can instantiate it with the proof delivered by `x`, yielding a proof of `B`.

Definition `tspec {A B} (f : ttac (A -> B)) (x : ttac A) : ttac B := '(b, gb) <- f; '(a, ga) <- x; ret (b a, gb ++ ga)`.

Dynamic goals in typed tactics.

Let us now turn towards the dynamic goals returned by typed tactics. To illustrate the usefulness of this component, we have changed the way `solve_tauto` works. In contrast to both previous versions, the typed-tactic version does not fail when the proposition cannot be (fully) proven.

¹ Recall that the `[m: ..]` syntax refers to Mtac2's universe-polymorphic list type (see Section 14.1).

² It is worth pointing out that typed tactics form an applicative functor [MP08].

Instead, it operates in a best-effort way, solving as many subgoals that come up during the traversal of the proposition as possible, but potentially leaving some open.

Best-effort strategies present a challenge to the typed tactics infrastructure: Whether a subgoal is solved or not depends on dynamic factors such as hypotheses introduced during the execution of the tactic, type class search, and other best-effort tactics being used in the implementation. The second component of `ttac` reconciles the static nature of typed tactics with such uncertainties by allowing typed tactics to optionally spawn dynamic goals.

The simplest such typed tactic is `demote`, which demotes a static goal to a dynamic one. The `demote` tactic is implemented in terms of `to_goal`, a helper function which takes care of creating one new dynamic goal from the current static goal. The code for both functions is given below.

```
Definition to_goal (A : Type) : M (A * goal) :=
  a <- evar A; ret (a, Metavar gs_active _ a).
```

```
Definition demote {A : Type} : ttac A :=
  '(a, g) <- to_goal A; ret (a, [m: g]).
```

To see how the best-effort strategy is reflected in the new implementation of `solve_tauto`, consider the disjunction case. When the tactic encounters a disjunction, it now only commits to either side if that side can be solved without any remaining subgoals. This is necessary because we no longer have a clear indicator of which side to choose due to potential remaining subgoals even in a successful execution of the solver. If neither branch can be solved this way, we demote the current static goal (Line 15) to a dynamic one.

Using an approach similar to that of `demote`, we can implement a typed tactic called `use` which applies a regular, dynamically-typed tactic to a static goal:

```
Definition use {A} (t : tactic) : ttac A :=
  '(a, g) <- to_goal A;
  gs <- t g;
  ret (a, gs).
```

An example of `use` can be found in Line 22, where we delegate the static goal of type `P` to the dynamically-typed `assumption` tactic. Note that `use` does not assert that the tactic `t` fully solves the static goal, *i.e.*, that it does not spawn new dynamic goals. A variant of `use`, called `by'`, enforces that assertion.³ It makes use of `filter_goals`, mentioned already in Section 10.3, for filtering out solved goals before asserting that the list of goals is empty. We present an example of `by'` in Section 12. Its implementation is given below.

```
1 Definition by' {A} (t : tactic) :=
2   '(m: a, g) <- to_goal A;
3   gs <- (t g >=> T.filter_goals);
4   match gs with
5   | [m:] => ret (m: a, [m:])
6   | _ => failwith "Goal not solved"
7   end.
```

³ It is called `by'` because `by` is a keyword in the Rocq grammar and cannot be used as the name of a definition.

Another very useful combinator for typed tactics is `TT.try`, which demotes the static goal in case its argument, another typed tactic, fails. The code⁴ of `TT.try` is given below. If demoting the goal is not desired, tactic developers can fall back to using `mtry` directly. Line 22 contains an example application of `TT.try`, which allows the goal `P` to remain unsolved when no matching assumption can be found.

```
Definition try {A} (t : ttac A) : ttac A :=
  mtry t with _ => demote : M _ end.
```

From dynamic to static goals.

Tactics such as `use` and `by'` form one side of static-dynamic interoperability and *demoting* goals is the key to achieving that. The other side, unsurprisingly, involves *promoting* dynamic goals to static ones—an operation that is fundamentally about reflecting dynamically acquired knowledge (e.g., by matching on the goal) into static knowledge. Accordingly, the key to promoting dynamic goals to static goals is a statically typed variant of `Mtac2`'s `match_goal` construct presented in Section 10.5 that reflects the knowledge about the goal type into the type of every branch.

The code of `solve_tauto` above does not make use of the new `match_goal` combinator, as it does not live at the boundary between typed and regular tactics. If, however, we wanted to lift `solve_tauto` into a regular tactic, we could easily do so by using `match_goal`. The code for this is given below. It matches the goal against any proposition `P` and then calls `solve_tauto P`.

```
Definition solve_tauto_regular : tactic :=
  (match_goal with
  | [[?(P : Prop) |- P] ] => solve_tauto P
  end)%TT.
```

Note that the branch of `match_goal` is statically typed: its type is `ttac P`. Calling a typed tactic that does not match that type will result in a typechecking error at definition time of the wrapper. Note, further, that both `match_goal` variants have type `tactic`. To differentiate between the two we use Rocq's scoping mechanism by wrapping the statically typed `match_goal` expression in the `TT` (short for typed tactic) scope.

While `solve_tauto_regular` nicely showcases the statically-typed variant of `match_goal`, it is arguably somewhat verbose. To avoid the boilerplate that comes with `match_goal`, we introduce a shorthand for cases in which only one statically-typed tactic is used inside a dynamically-typed one. This shorthand is called `to_T` and its code is given in Figure 11.2.

The `to_T` function takes a statically-typed tactic `F` of type `A` and attempts to unify the current goal, bound to variable `B`, with that type `A`. For this, we use cumulative unification (see Section 13.4.3), which returns an injection (`g`) from `A` to `B` if it succeeds. `g` is guaranteed to be the identity function, i.e., `(fun x => x)`. We apply it to the result of running `F`—a proof `a` of type `A`—and reduce away the identity function with one step of beta reduction to avoid cluttering our proof terms with such superfluous applications. This then yields the desired proof of the goal type `B`.

⁴ The type annotation `(. . : M _)` is only included to help Rocq's typechecker and can be safely ignored by the reader.

Figure 11.2: The code of `to_T`, a shorthand for applying statically-typed tactics inside dynamically-typed ones.

```

1 Definition WrongType : Exception. constructor. Qed.
2 Definition to_T {A} (F : ttac A) : tactic :=
3   match_goal with
4   | [[? B |- B]] =>
5     og <- M.unify_univ A B UniCoq;
6     match og with
7     | mSome g =>
8       '(a, gs) <- F;
9       let ga := reduce (RedOneStep [RedBeta]) (g a) in
10      ret (ga, gs)
11     | mNone => raise WrongType
12   end
13 end.

```

With `to_T`, we can redefine `solve_tauto_regular` without `match_goal`:

```

1 Definition solve_tauto_regular : tactic :=
2   P <- evar Prop; to_T (solve_tauto P).

```

Type class search in typed tactics.

In addition to all the typed tactics shown above, our case study makes use of another typed tactic, `TT.apply_`, which triggers type class search on the current goal.⁵ We saw an example of a type class, `PureExec`, in Section 8. A type class instance of `PureExec` φ `e1` `e2` signifies that the expression `e1` reduces to `e2` given that sufficient precondition φ holds. In Section 12.3, we will use `TT.apply_` to search for instances of `PureExec` for a given candidate redex `e1`.

⁵ The name `apply_` is a reference to Ltac's way of triggering type class search: `apply -`.

12

Case study: Iris Proof Mode

In this section we return to the Iris Proof Mode (IPM) case study that we introduced in Section 8.1. Before showing how we can use Mtac2 to build more robust tactics that address the challenges we posed, we give a brief introduction to IPM.

The core feature of IPM is that it enables one to carry out separation logic proofs with the look and feel of proofs in Rocq. Let us take a look at a basic proof of a simple lemma in separation logic:

```
1 Lemma sep_exist A (P R : uPred M) (Ψ : A → uPred M) :
2   P * (∃ a, Ψ a) * R → ∃ a, Ψ a * P.
3 Proof.
4   iIntros "[HP [HPsi HR]]". iDestruct "HPsi" as (x) "HPsi".
5   iExists x. iSplitL "HPsi"; iAssumption.
6 Qed.
```

This lemma looks like an ordinary lemma in Rocq; the crucial difference is that we are using the connectives of the separation logic (separating conjunction $*$ and magic wand $\multimap$). After calling the first tactic, we end up with the following goal:

```
1 A : Type
2 P, R : uPred M
3 Ψ : A → uPred M
4 ----- (1/1)
5 "HP" : P
6 "HPsi" : ∃ a, Ψ a
7 "HR" : R
8 -----*
9 Ψ x * P
```

As we see here, an additional context appeared that contains the hypotheses in separation logic. The tactics of IPM behave much like the ordinary Rocq tactics, but operate on the separation logic context. For example, after calling `iDestruct "HPsi" as (x) "HPsi"`, the hypothesis `"HPsi"` will be turned into `Ψ x` and `x` will be added to the Rocq context.

Now, let us take a look at what is going on. IPM uses a deeply embedded representation of the contexts of separation logic `envs`, and the above goal is just a pretty printed version of the entailment relation `envs_entails`:

```

1 Record envs (M : ucmraT) := Envs
2   { env_persistent : env (uPred M);
3     env_spatial : env (uPred M) }.
4 Definition envs_entails {M} : envs M -> uPred M -> Prop := ...

```

[Jun+18b] Jung et al., “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”

The type `env A` represents association lists with values of type `A` and strings as keys. The type `ucmraT` represents user-chosen ghost state—a parameter necessary to instantiate Iris—and the type `uPred M` is the type of Iris propositions, which depends on that ghost state [Jun+18b].

Although the example above displayed just the *spatial context* `env_spatial`, there is in fact another context `env_persistent` for the so called *persistent* hypotheses. For those readers not familiar with Iris, this context can be safely ignored for the purpose of this chapter.

12.1 SYMBOLIC EXECUTION IN LTAC

Apart from providing tactics for manipulating the logical connectives of separation logic, IPM offers a set of tactics for symbolically executing program steps. These tactics provide a great deal of convenience to IPM users by taking care of applying the appropriate proof rule, automatically selecting required hypotheses, and introducing new ones, as well as simplifying and reshaping the remaining goals. In the introduction we already saw `wp_pure`—a symbolic execution tactic for pure reduction steps. IPM also supplies tactics for heap operations: `wp_load`, `wp_store`, `wp_cas`, etc.

IPM tactics generally rely on a set of helper tactics and type classes and instances thereof. In the following, we first give a detailed account of the various bits and pieces that go into the `wp_pure` tactic for symbolic execution and then explain how to port it to `Mtac2`.

Figure 12.1: The type of Lemma `tac_wp_pure`; Ltac code for `wp_pure`. (Unchanged from Figure 8.1.)

```

1 Lemma tac_wp_pure `{heapG Σ} Δ s E e1 e2 φ Φ :
2   PureExec φ e1 e2 ->
3   φ ->
4   envs_entails Δ (wp s E e2 Φ) ->
5   envs_entails Δ (wp s E e1 Φ).
6
7 Tactic Notation "wp_pure" open_constr(epat) :=
8   lazymatch goal with
9   | |- envs_entails _ (wp ?s ?E ?e ?Φ) =>
10     let e := eval simpl in e in
11     reshape_expr e ltac:(fun K e' =>
12       unify e' epat;
13       eapply (tac_wp_pure _ _ _ (fill K e'));
14       [apply _ (* PureExec *)
15         |try fast_done (* The pure condition for PureExec *)
16         |wp_expr_simpl_subst; try wp_value_head (* new goal *)
17       ])
18   || fail "wp_pure: cannot find" epat "in" e
19     "or" epat "is not a reduct"
20   | _ => fail "wp_pure: not a 'wp'"
21 end.

```

Figure 12.1 (from Line 7 onward) contains the Ltac code of the `wp_pure` tactic. `wp_pure` starts by examining the goal: Symbolic execution only applies to weakest precondition IPM goals. If the tactic finds such a goal, it computes a simplified version of the expression therein and hands that to another helper tactic called `reshape_expr`. The abridged code of `reshape_expr` is given in Figure 12.2. Its job is to 1) iteratively compute all possible decompositions of the given expression into evaluation contexts and redex, and 2) call the continuation given to it with each decomposition until the continuation succeeds. We will now explain how this decomposition into evaluation contexts works.

```

1 Ltac reshape_expr e tac := let rec go K e :=
2   match e with
3   | _ => tac K e
4   | App ?e1 ?e2 =>
5     reshape_val e1 ltac:(fun v1 => go (AppRCtx v1 :: K) e2)
6   | App ?e1 ?e2 => go (AppLCtx e2 :: K) e1
7   | ...
8   end in go (@nil ectx_item) e.

```

Figure 12.2: The `reshape_expr` Ltac tactic, showing only the representative application case.

Evaluation contexts.

The language used in the default instantiation of Iris, `heap_lang`, is a lambda calculus with sums, products, and references. Its operational semantics is formalized using evaluation contexts [FH92]—expressions with a hole for the expression in evaluation position. The mechanization of `heap_lang` deviates from the standard presentation of evaluation contexts by representing them as lists of evaluation context *items*. The list structure reflects the recursive nature of evaluation contexts: They are linear trees. The development includes a plugging function, called `fill`, which inserts an expression into the hole of an evaluation context to form a closed expression. Due to the use of evaluation context *items*, `fill` is defined as a fold over a plugging function on those items. That function is called `fill_item`. The (abridged) definition of evaluation contexts and the code of `fill_item` are given in Figure 12.3.

[FH92] Felleisen et al., “The revised report on the syntactic theories of sequential control and state”

```

Inductive ectx_item :=
| AppLCtx (e2 : expr)
| AppRCtx (v1 : val)
| ...
Definition fill_item (Ki : ectx_item) (e : expr) :=
  match Ki with
  | AppLCtx e2 => App e e2
  | AppRCtx v1 => App (of_val v1) e
  | ...
  end.

```

Figure 12.3: Evaluation contexts and the `fill_item` function (reduced to the representative application case).

The purpose of using `reshape_expr` becomes clear once we turn our attention to the continuation provided by `wp_pure`. The call to `unify` in Line 12 accepts only those redexes `e'` that match the pattern `epat` given

to `wp_pure` by the user. If the match does not succeed, Ltac’s backtracking mechanism ensures that `reshape_expr` calls the continuation with the next decomposition. If the redex `e'` indeed matches `epat`, `wp_pure` applies `tac_wp_pure`, whose type is shown in Figure 12.1. The lemma serves as glue between `wp_pure` and a type class—`PureExec`—used to express sufficient conditions for pure reduction steps. It witnesses the fact that, given such a sufficient condition φ to reduce `e1` to `e2`, it suffices to prove φ and continue the proof with `e2`.

The remaining code of `wp_pure` deals with the subgoals spawned by applying `tac_wp_pure`. A call to type class search (`apply _`) solves the first subgoal: an instance of `PureExec`, establishing the sufficient condition φ for the pure reduction step. That condition, in turn, is taken care of on a best-effort basis by an IPM variant of Rocq’s `done` tactic, `fast_done`. Then, `wp_pure` attempts to call its last helper tactic, `wp_value_head`, whose job it is to check if the current expression is fully reduced to a value. If that is the case, it suffices to show the postcondition Φ for that value and the goal is transformed accordingly using beta reduction—`lazy beta`—to substitute the value in Φ . The code of both helper tactics and the lemmas supporting them is given below.

```

1 Lemma tac_wp_value `{heapG Σ} Δ s E Φ e v :
2   IntoVal e v ->
3   envs_entails Δ (Φ v) -> envs_entails Δ (WP e @ s; E {{ Φ }}).
4 Ltac wp_value_head := eapply tac_wp_value; [apply _|lazy beta].

1 Lemma tac_wp_expr_eval `{heapG Σ} Δ s E Φ e e' :
2   e = e' ->
3   envs_entails Σ (WP e' @ s; E {{ Φ }}) ->
4   envs_entails Δ (WP e @ s; E {{ Φ }}).
5 Ltac wp_expr_simpl_subst t :=
6   try (eapply tac_wp_expr_eval; [simpl_subst; reflexivity|]).

```

12.2 PROBLEMS WITH THE LTAC IMPLEMENTATION

The Ltac implementation of `wp_pure` suffers from several problems:

Goal-Lemma Mismatch. While `wp_pure` and its helper tactics do make use of Ltac’s `lazymatch` to inspect the shape of the goal, Ltac offers no way of statically exploiting this knowledge to ensure that the conclusion of lemmas applied with `eapply` indeed match the shape of the goal. A mistake here will lead to a failure at execution time of the tactic.

Lemma-Arguments Mismatch. Ltac offers no static typechecking. Thus, even if we use the right lemmas, we cannot statically ensure that we invoke them with arguments of the right type, or even with the right number of arguments. A mistake here will manifest itself only during the execution of the tactic.

Subgoal-Tactic Mismatch. Similarly, there is no guarantee that follow-up tactics—even highly specific ones such as `wp_value_head`—can be applied to the subgoals they were assigned to. Again, developers (and

their users) will only find out about such mistakes when executing the tactic.

Non-Modular Error Handling. Due to the lack of modular error handling in Ltac, any such runtime failure will cause `reshape_expr` to backtrack and try different decompositions of the expression—a behavior meant to be invoked only when the decomposition does not yield an expression in evaluation position for which a `PureExec` instance exists.

We address all of these issues in the Mtac2 port of `wp_pure` described below.

12.3 SYMBOLIC EXECUTION IN MTAC2

The code of the Mtac2 version of `wp_pure`—now a typed tactic—is given in Figure 12.4. The code follows the structure of the Ltac version. The remainder of this section addresses the differences between both versions and explains how the Mtac2 version resolves the issues we identified with the Ltac version. In particular, in Section 12.3.1, we show `reshape_expr` in Mtac2 (Line 4), and explain the new backtracking protocol driven by the `TryNextDecomposition` exception (Lines 6, 10). In Section 12.3.2, we show the Mtac2 implementation of `wp_expr_simpl_subst` and `wp_value_head`. Finally, in Section 12.3.3, we briefly describe how `wp_pure` can be converted into a regular, dynamically-typed tactic.

```

1 Definition wp_pure `{heapG Σ} {Δ s e E Φ} (epat : expr):
2   ttac (envs_entails Δ (WP e @ s ; E {{ Φ }})) :=
3   let e := rsimpl e in
4   mtry reshape_expr_wp e (fun K e' =>
5     mtry unify_or_fail UniEvarconv e' epat
6     with NotUnifiable e' epat => raise TryNextDecomposition end;;
7     `e2 φ <- evar _;
8     TT.apply (tac_wp_pure _ _ _ (fill K e') e2 φ _)
9     <*> (mtry TT.apply_ with (* PureExec *)
10       | _ => raise TryNextDecomposition
11       end)
12     <*> TT.try (T.try fast_done) (* Pure condition *)
13     <*> `e' <- evar _; (* new goal *)
14     wp_expr_simpl_subst e'
15     <*> TT.try wp_value_head)
16 with ReshapeExprExc =>
17   mfail "wp_pure: cannot find " epat " in " e
18   " or " epat " is not a reduct"
19 end.
```

Figure 12.4: The Mtac2 version of `wp_pure`.

12.3.1 reshape_expr

The `reshape_expr` tactic is the backbone of `wp_pure`, performing the heavy lifting of decomposing expressions and also backtracking if the current decomposition is not suitable. In Mtac2, `reshape_expr` takes on an additional role on the level of types: refining the static goal of its continuation

to depend not on the original expression e but instead on the decomposition $\text{fill } K \ e'$. This is the key ingredient to wp_pure 's well-typedness, which crucially relies on the application of tac_wp_pure (Line 8 of Figure 12.4) to $\text{fill } K \ e'$. Thus, reshape_expr plays an important role in solving the **Goal-Lemma Mismatch** issue.

Figure 12.5: The Mtac2 version of reshape_expr , showing the same cases displayed in the Ltac version in Figure 12.2 and, additionally, an explicit branch for all cases intended to fail.

```

1 Definition ReshapeExprExc : Exception. constructor. Qed.
2 Definition TryNextDecomposition : Exception. constructor. Qed.
3 Definition reshape_expr (e : expr) {B : expr -> Type}
4   (tac : forall K e', M (B (fill K e')))) : M (B e) :=
5   (mfix2 go (K : list ertx_item) (e' : expr) :
6     M (B (fill K e')) :=
7     mtry tac K e' with TryNextDecomposition =>
8     match e' as e' return M (B (fill K e')) with
9     | App e1 e2 =>
10       mtry reshape_val e1 (fun v1 => go (AppRCtx v1 :: K) e2)
11       with ReshapeExprExc => go (AppLCtx e2 :: K) e1 end
12     | ...
13     | Var _ | Lit _ | Fork _ | Rec _ _ _ => raise ReshapeExprExc
14     end
15   end) [] e.
16 Definition reshape_expr_wp `{irisG heap_lang Σ} {s E Φ}
17   (e : expr)
18   (tac : forall K e',
19     ttac (envs_entails Δ (WP fill K e' @ s ; E {{ Φ }}))) :
20   ttac (envs_entails Δ (WP e @ s ; E {{ Φ }})) :=
21   reshape_expr e (B:=fun e =>
22     envs_entails _ (WP e @ _ ; _ {{ _ }}) *m _
23   ) tac.

```

The Mtac2 code of reshape_expr is shown in Figure 12.5. The entire tactic is statically typechecked. Note, though, that we do not restrict the tactic to work on IPM goals (weakest preconditions), which reduces visual clutter quite significantly. In Line 16, we provide an instantiation of reshape_expr , called reshape_expr_wp , which has the more specific type relevant to the application of this tactic in wp_pure . In addition to its well-typedness, reshape_expr is also guaranteed to exhaustively cover all constructors of the expr type. This is enforced by our use of Rocq's `match` construct. Indeed, Rocq forces us to add a branch for all expressions we do not intend to decompose (Line 13).

The type of reshape_expr witnesses the tactic's correctness. Its type promises to prove any property B of expression e while the continuation tac only provides proofs of B for expressions of the form $\text{fill } K \ e'$. Thus, to call tac in Line 7 with a decomposition K and e' , we need to ensure that $\text{fill } K \ e'$ is equal to e . This condition is initially true because $\text{fill } [] \ e = e$. At every recursive call, we have to pass a decomposition of $\text{fill } K \ e'$ into a new evaluation context K' and expression e'' such that $\text{fill } K' \ e''$ is still equal to the previous decomposition $\text{fill } K \ e'$. If we fail to do so, the typechecker will not accept our definition. Thus, we ensure that the continuation tac is always called with a valid decomposition of e .

Let us take a closer look at the case for application of e_1 to e_2 in Line 9. First, in Line 10, we attempt to reshape e_1 into a value.

This is done by calling another helper tactic, `reshape_val`. `reshape_val`, similarly to `reshape_expr`, certifies in its type that the value `v1` given to its continuation is equal to `e1` modulo conversion to an expression. If `reshape_val` succeeds, we push the evaluation context (`e1 o`)—represented by `AppRCtx v1`—onto the list of evaluation contexts and recurse with `e2` as the decomposition candidate. If `reshape_val` fails to find a value equivalent to `e1` (which it signals by raising `ReshapeExprExc`), we instead proceed by extending the list of evaluation contexts by (`o e2`), i.e., `AppLCtx e2`, and recurse with `e1` as the decomposition candidate.

In addition to the benefits mentioned above, the `Mtac2` version of `reshape_expr` puts its clients in full control of backtracking. To this end, we introduce a new exception, `TryNextDecomposition`. Only if the client raises this exception does `reshape_expr` backtrack. The other exception, `ReshapeExprExc`, is used solely to signal actual failure. This strict separation solves the issue of **Non-Modular Error Handling** from the `Ltac` version.

12.3.2 `wp_simpl_subst` and `wp_value_head`

Since the tactic `wp_expr_simpl_subst` applies the lemma `tac_wp_expr_eval` directly, it lends itself very well to conversion into a typed tactic. It is worth noting that `wp_expr_simpl_subst` represents a *partial application* of `tac_wp_expr_eval`: It only attempts to prove the first hypothesis of the lemma. The remaining hypothesis is a static subgoal which we can easily represent in the type of the tactic.

```
1 Definition wp_expr_eval `{hG : heapG Σ} {Δ s E Φ e} e' :
2   ttac (envs_entails Δ (WP e' @ s ; E {{ Φ }}) ->
3     envs_entails Δ (WP e @ s ; E {{ Φ }}})) :=
4   TT.apply (tac_wp_expr_eval Δ s E Φ e e')
5   <*> (TT.use (simpl_subst;; T.reflexivity)).
```

Similarly, `wp_value_head` is a very thin layer around `tac_wp_value_head` and, thus, very easily converted to a typed tactic. As `wp_value_head` is the very last tactic applied by `wp_pure` (and similar tactics), we use `TT.try` to return any unsolved subgoal as a dynamic goal after simplifying any beta redexes.

```
1 Definition wp_value_head `{heapG Σ} {Δ} {s} {E} {e} {Φ} :
2   ttac (envs_entails Δ (WP e @ s ; E {{ v, Φ v }}})) :=
3   e' <- evar _;
4   TT.apply (tac_wp_value Δ s E Φ e e')
5   <*> TT.apply_ <*> TT.use (T.treduce (RedStrong [rl: RedBeta])).
```

Both tactics have very informative static types which we can exploit to statically ensure the absence of any **Subgoal-Tactic Mismatch**.

Note that the `simpl_subst` tactic used by `wp_expr_simpl_subst` has also been ported to `Mtac2`. We ask the interested reader to peruse the `Rocq` development for further details.

12.3.3 A dynamically-typed tactic for `wp_pure`

The `Mtac2` version of `wp_pure` presented in Figure 12.4 is a typed tactic whose type documents exactly which goals the tactic can be applied to.

Users of IPM, however, may not need the static types of `wp_pure` and may also want to easily call the tactic *without* having to specify all its arguments. To this end, we define a tactic wrapper around `wp_pure`, which uses the statically typed `match_goal` construct to convert the dynamic goal to the static goal required by `wp_pure`. The code for this wrapper is given below:

```

1 Definition wp_pure_wrap (epat : expr) : tactic :=
2   (match_goal with
3     | [[?  $\Sigma$  (hG : heapG  $\Sigma$ )  $\Delta$  s E e  $\Phi$ 
4       |- envs_entails  $\Delta$  (WP e @ s ; E {{  $\Phi$  }})] ] =>
5     wp_pure epat
6     | [[?G |- G ] ] =>
7       mfail "Goal does not match an IPM weakest precondition."
8   end)%TT.

```

12.4 BENEFITS OF THE MTAC2 IMPLEMENTATION

In the previous sections, we identified various components in the Mtac2 version of `wp_pure` that contributed to resolving the issues we identified with the Ltac version in Section 12.2. We summarize these points here:

Goal-Lemma Mismatch The Mtac2 version of `wp_pure` takes full advantage of the strong types given to its helper tactics. The dependently-typed tactic `reshape_expr` refines the static goal of the innermost continuation to

```
ttac (@envs_entails (iResUR  $\Sigma$ )  $\Delta$  (@wp iG s E (fill K e')  $\Phi$ ))
```

Rocq is then able to statically match the type of the lemma being applied in `wp_pure`, `tac_wp_pure`, to this static goal—ensuring that any mismatch between these two types is reported at *definition* time of the tactic.

Lemma-Arguments Mismatch The Mtac2 version of `wp_pure` makes use of the statically-typed `TT.apply` tactic (Line 8). To be able to give a static type to the application of `tac_wp_pure`, we first introduce metavariables to fully instantiate the lemma (Line 7). Once introduced, these variables suffice to statically type the application of the lemma, and, thus, to ensure the absence of any mismatches between the lemma’s type and its arguments at *definition* time.

Subgoal-Tactic Mismatch The types given to `wp_expr_simpl_subst` and `wp_value_head` are informative enough to ensure that they can indeed be applied to the subgoal corresponding to `tac_wp_pure`’s last hypothesis. The tactics used for the two preceding subgoals—type class search via `TT.apply_` and the best-effort `fast_done` tactic—do not have interesting types: They claim to prove, or at least attempt to prove, any goal given to them.

Non-Modular Error Handling To make use of the new backtracking mechanism in the Mtac2 version of `reshape_expr`, `wp_pure` throws the designated backtracking exception `TryNextDecomposition` if 1) the redex `e'` does not match the pattern (`epat`) specified by the user, or 2) no sufficient condition was found for the reduction of `e'`. This fixes the erratic backtracking behavior that the original Ltac version may exhibit.

Before concluding the `wp_pure` case study we would like to point out that the extra work that types impose is quickly amortized, since most of the symbolic execution tactics of Iris follow the same pattern as `wp_pure` and can therefore reuse the abstractions defined here. In fact, in the following subsection, we present additional tactics that make use of the same building blocks as `wp_pure`.

12.5 SYMBOLIC EXECUTION OF HEAP OPERATIONS

We have used the infrastructure introduced for `wp_pure` to port two additional symbolic execution tactics to Mtac2: `wp_load` and `wp_store`, which symbolically execute read and write operations on the heap. In order to avoid repeating material from the `wp_pure` case study, we will discuss these tactics at a very high level. Our goal is not to perform a detailed comparison between their Mtac2 ports and the original Ltac versions. Instead, we want to demonstrate that the approach taken for porting `wp_pure` (i.e., making use of statically-typed tactics), scales to more complex IPM tactics. In particular, these additional case studies showcase how the statically-typed approach fares when some type information is only available at execution time.

Interacting with the heap.

In contrast to `wp_pure`, which does not interact with the separation logic context, `wp_load` and `wp_store` rely on the presence of a particular separation logic assumption in the context: the *points-to* assertion, written $x \mapsto v$ (“location x points to value v ”). It signifies ownership of the location x and guarantees that x has value v . There exists a *fractional* version, written $x \mapsto_q v$ which denotes partial ownership of x with $q \leq 1$. In separation logic, the points-to assertion is a necessary assumption for interacting with the respective location in memory. This is captured by lemmas `load_spec` and `store_spec` below. Note that reading only requires partial ownership whereas writing always requires full ownership.

```
1 Lemma load_spec s E x q v :
2   x  $\mapsto_q$  v  $\text{--}^*$  WP (Load (Lit (LitLoc x))) @ s; E {{ v, x  $\mapsto_q$  v }}.
3 Lemma store_spec s E x v' v :
4   x  $\mapsto$  v'  $\text{--}^*$  WP (Store (Lit (LitLoc x)) v) @ s; E {{ _, x  $\mapsto$  v }}.
```

`load_spec` states that owning a part of location x and knowing its value to be v guarantees that reading x indeed yields v . We regain our partial ownership at the original value v . For `store_spec`, the initial value v' of

x does not matter when we write a new value v to it. All we need is full ownership of x . After executing the write operation, we regain full ownership of x , now with the new value v .

Automation for heap operations.

To facilitate reasoning about heap operations, IPM provides two tactics called `wp_load` and `wp_store`. They provide symbolic execution for reads and writes. In particular, the tactics automatically select a points-to assertion from the spatial context matching the next load or store operation to be executed, apply the corresponding lemma (`load_spec` or `store_spec`), and re-introduce the resulting points-to assertion.

For example, consider a proof of a weakest precondition for an expression with a write operation in evaluation position. The initial goal looks as follows:

```

1 ...
2 "Hx" : x ↦ v'
3 -----*
4 WP ... (Store (Lit (LitLoc x)) v) ... {{ ... }}
```

An application of the `wp_store` tactic changes the goal to the following, in which a) the value of x has been replaced by v in the points-to assertion `"Hx"`, and b) the `Store` operation has been replaced by its result, the unit literal written `Lit LitUnit`.

```

1 ...
2 "Hx" : x ↦ v
3 -----*
4 WP ... (Lit LitUnit) ... {{ ... }}
```

In the remainder of this section, we give a brief overview of how `wp_load` and `wp_store` work and then proceed to show the `Mtac2` versions.

Symbolic execution for heap operations in Ltac.

From here on, we focus on `wp_store` as it showcases more interesting interaction with the separation logic context than `wp_load`. The Ltac code of IPM's `wp_store` tactic is shown in Figure 12.6. As with `wp_pure` (Section 12), the `wp_store` tactic relies on an auxiliary lemma, called `tac_wp_store`, which is presented in the same figure. Owing to the necessity of interacting with the spatial context of IPM, `tac_wp_store` is more complex than `tac_wp_pure`. Its type expresses both the change to the goal (reduction of the `Store` expression) as well as the interaction with and the changes to the spatial context. The latter is expressed by lines 3 through 5:

1. Line 3 proves that the points-to assertion $x \mapsto v'$ can be found in the original context Δ under the identifier i (`"Hx"` in our example).
2. Line 5 ensures that the new context Δ' can be obtained from the original context Δ by replacing the assumption i by a new assumption of the same name with updated type $x \mapsto v$.

```

1 Lemma tac_wp_store  $\Delta$   $\Delta'$  s E i K x v' e v  $\Phi$  :
2   IntoVal e v ->
3   envs_lookup i  $\Delta$  = Some (x  $\mapsto$  v) ->
4   envs_simple_replace i false (Esnoc Enil i (x  $\mapsto$  v))  $\Delta$  =
5     Some  $\Delta'$  ->
6   envs_entails  $\Delta'$  (WP fill K (Lit LitUnit) @ s; E {{  $\Phi$  }}) ->
7   envs_entails  $\Delta$ 
8     (WP fill K (Store (Lit (LitLoc x)) e) @ s; E {{  $\Phi$  }}).
9
10 Tactic Notation "wp_store" :=
11   iStartProof;
12   lazy_match goal with
13   | |- envs_entails _ (wp ?s ?E ?e ?Q) =>
14     first
15       [reshape_expr e ltac:(fun K e' =>
16         eapply (tac_wp_store _ _ _ _ K))
17         | fail 1 "wp_store: cannot find 'Store' in" e];
18     [apply _
19       | let l := match goal with
20         | _ = Some (?l  $\mapsto$  { _ } _) => l
21         end
22       in
23         iAssumptionCore
24         || fail "wp_store: cannot find" x " " ?"
25         | env_cbv; reflexivity
26         | wp_expr_simpl;
27         try first [wp_pure (Seq (Lit LitUnit) _) | wp_value_head]]
28   | _ => fail "wp_store: not a 'wp'"
29   end.

```

Figure 12.6: The Ltac version of wp_store.

The `wp_store` tactic uses the `reshape_expr` tactic from Section 12 to search for a `Store` operation in evaluation position. There is no explicit unification of the candidate redex `e'` with `Store ...`; instead, this happens implicitly by applying `tac_wp_store` (Line 16). If the application succeeds, `wp_store` proceeds to handle the remaining subgoals *outside* of `reshape_expr`. This is done to avoid superfluous backtracking in `reshape_expr` when follow-up tactics fail. As there is only ever one candidate redex for a `Store` operation there is no need for `reshape_expr` to backtrack. For the same reason, `wp_store` uses `fail 1` (Line 17) to fail *over* `reshape_expr` when the application of `tac_wp_store` fails.

After applying lemma `tac_wp_store`, we are left with four subgoals, the first of which corresponds to `tac_wp_store`'s `IntoVal` assumption and is immediately discharged by type class search via `apply_` (Line 18). The `IntoVal` type class is used to make `tac_wp_store` applicable in cases where the expression to be stored, `e`, is not yet *syntactically* a literal value.

The second subgoal (Lines 19 through 24) is a proof that $x \mapsto v$ can be found in context Δ under identifier `i`. To ensure that the following subgoal (where the hypothesis `i` is replaced) can be solved, we need to be certain that the identifier `i` will be unified with a concrete value. The way this is achieved by the `iAssumptionCore` tactic—while certainly interesting in its own right—is beyond the scope of this thesis.

The third subgoal (Line 25) corresponds to the computation of the new context Δ' . Δ' is computed by the Rocq function `envs_simple_replace`.

As the goal is stated as an equality it is solved by a call to call-by-value reduction (`env_cbv`), which is a restricted reduction tactic that reduces only computation relevant to the (new) context, but not the context's contents. A final call to `reflexivity` will force the as-of-yet undetermined existential variable Δ' to be unified with the result of the computation.

The last subgoal (Lines 26 through 27), corresponding to the remaining weakest precondition proof, is simplified using the `wp_expr_simpl` tactic introduced in Section 12. In addition, `wp_pure` is called on the resulting subgoal to look for a sequential composition (`Seq`) that might be trivially reducible now. Finally, as with `wp_pure`, the `wp_value_head` tactic will simplify the goal to an application of the post condition if and only if the remaining expression is in fact a value.

Symbolic execution for heap operations in Mtac2.

The Mtac2 version of `wp_store` is shown in Figure 12.7. There are a number of differences to the Mtac2 version of `wp_pure`—mostly owed to `wp_store`'s complexity—and we expand on these below. We will also briefly discuss noteworthy differences to the original Ltac implementation of `wp_store`.

Figure 12.7: The Mtac2 version of `wp_store`.

```

1 Definition wp_store {Δ s e E Φ} :
2   ttac (Δ ⊢ WP e @ s ; E {{ Φ }}) :=
3   reshape_expr_wp e (fun K e' =>
4     `Δ' Δ' v' v i l <- M.evar _;
5     TT.change_dep (fun e => _ ⊢ (WP e @ _ ; _ {{ _ }})) _ (
6       TT.apply (tac_wp_store Δ Δ' s E i K x v' e' v Φ)
7       <*> TT.apply_ (* IntoVal *)
8       <*> (iAssumptionCore
9         || [NotFoundInEnv] mfail "Cannot find " l " → ?")
10      <*> TT.use (env_cbv;; T.reflexivity)
11      <*> `e' <- evar _; wp_expr_simpl_subst e'
12      <*> first [
13        `e' <- evar _;
14        TT.change_dep (fun e => _ ⊢ WP e @ _ ; _ {{ _ }}) e'
15        (wp_pure (Seq (Lit LitUnit) e'))
16        | TT.try wp_value_head]
17    ) || [TT.TTchange_Exception] raise TryNextDecomposition
18  ) || [ReshapeExprExc] mfail "wp_store: cannot find 'Store' in " e.

```

Before we compare the code to Ltac or other Mtac2 tactics, we need to introduce the syntactic sugar used to reduce the visual complexity of the types involved in `wp_store`. In particular, $\Delta \vdash P$ is shorthand for `envs_entails Δ P`. Additionally, we use novel syntax to reduce the syntactic noise of catching specific exception: `f || [e1 .. en] g` attempts to execute tactic `f` and handles exceptions `e1` through `en`—but not other exception—by executing `g`. Other exceptions are simply rethrown. This syntax has purposefully been made very similar to the `||` operator of dynamically-typed tactics in Mtac2, allowing for similar compactness when writing dynamically-typed and statically-typed tactics. We use this syntax in Lines 9, 17, and 18 to handle expected failures and, implicitly, treat all other failures as bugs by allowing them to escape and eventually

reach the user (or the enclosing tactic).

Comparing the Mtac2 versions of `wp_store` and `wp_pure`, we see one immediate difference: `wp_store` does not directly apply `tac_wp_store` in the continuation given to `reshape_expr_wp`. Instead, we first call `TT.change_dep` (Line 5), a typed tactic that dynamically asserts the shape of the goal to be compatible with the typed tactic specified as its last argument (the application of `wp_store`, in our case). The reason for this is a mismatch between the static type guaranteed by `reshape_expr_wp` and that required by `tac_wp_store`: `tac_wp_store` only applies to goals in which a store operation is in evaluation position; but `reshape_expr_wp` only guarantees to produce *some* expression in evaluation position.

With `TT.change_dep`, whose type is shown below, we can quickly and precisely pinpoint this discrepancy and ask for a dynamic check to reconcile the types. We do this by specifying, in `change_dep`'s first argument—the predicate `B`—exactly which part of the goal to abstract over. In our case, we abstract over the expression in evaluation position. To the outside, `TT.change_dep` will have a type that solves a weakest precondition goal with *any* expression in evaluation position. The tactic given to it, however, only solves those goals with store expressions in evaluation position.

```
Definition change_dep {X : Tye} (B : X -> Type)
  (x : X) {y : X} (f : ttac (B x)) : ttac (B y).
```

We use `TT.change_dep` a second time before applying `wp_pure` in Line 14 in order to reduce away sequential compositions left after reducing `Store` to the unit value.

The remaining structure of `wp_store` is very similar to `wp_pure` in that it handles all subgoals arising from the application of `tac_wp_store` inside the continuation given to `reshape_expr`. This is in contrast to the Ltac version of `wp_store`, which has to explicitly opt out of `reshape_expr`'s backtracking by moving all the followup tactics outside of the continuation. The Mtac2 version does not need to do that because the Mtac2 port of `reshape_expr_wp` only backtracks when a specific exception is thrown (Section 12.3.1). Apart from that, we mirror the general strategy of the original `wp_store` Ltac code very closely.

Concluding remarks.

Without going into further details about the various helper tactics involved in `wp_store` (and `wp_load`), we can already see that statically-typed tactics scale to this more complex setting and allow us to port and improve the original Ltac code. In particular, We have seen how a lack of static type information can be overcome by adding very precise dynamic checks (such as `change_dep`).

13

Operational semantics of Mtac2

In this section we develop a formal account of Mtac2. When it comes to the semantics, there are a few salient aspects of Mtac2 —relative to Mtac1— that are of particular interest. We summarized them here.

Backtracking.

As we have already seen in Section 10.5, exposing backtracking via `mtry` allows us to construct `mmatch`—formerly a primitive—directly in Rocq. We consider this change an example of an overarching design decision in Mtac2: to provide small, orthogonal primitives that can be combined to form more complex meta-programs in Rocq. While this arguably is a satisfying way to design semantics from a theoretical standpoint, we also find constructing and, somewhat more crucially, maintaining statically typed Rocq code a much more reliable process than maintaining OCaml code that performs untyped AST manipulation.

Non-blocking primitives.

A major change in Mtac’s semantics is the new take on invalid input: instead of aborting the execution of the Mtac program, Mtac2 throws a meaningful exception. This change, as will be explain further below, was necessary to control the execution of an Mtac program in situations where the error is not necessarily an unsolvable one.

Reduction of terms.

The reduction mechanism for terms changed to use a clever trick enabling a reduced term to appear equal to the typechecker to the original term. This change was crucial to avoid unnecessary coercions in the code of tactics.

Extended set of primitives.

Mtac2 not only has more primitive primitives, as discussed above, but also presents an extended set of primitives. For instance, Mtac1 only allowed the construction of a λ -abstraction (the `abs` primitive), while

Mtac2 also includes the construction of the other binders in the theory (`let-in`, `forall`, `fix`).

Virtual machine.

The interpreter of Mtac1 was implemented somewhat naively, performing badly in large examples. Instead, in Mtac2 we implemented it as a virtual machine. This change also has an impact in the style of the rules we will see in this section, as they follow closely the implementation of the VM.

The evaluation of an Mtac2 term of type $\mathbb{M} \ A$ has three possible outcomes:

- a) it terminates successfully returning a term `ret e` where e has type A ; or
- b) it terminates by throwing an exception `raise e`, where e has type `Exception`; Or
- c) it diverges.

Hence, the set of values is defined as:

Definition 13.0.1 (Values). $v \in \text{Values} ::= \text{ret } e \mid \text{raise } e$.

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

Note that this differs from Mtac1 [Zil+15] in that Mtac1 also allowed for the possibility that the execution of a tactic could get *blocked*. This would arise, for instance, if the tactic tried to leak a variable introduced by `nu` or attempted to pattern match on a variable. In effect, Mtac1’s execution would “panic” (i.e., throw an uncatchable exception) upon encountering these types of errors. In contrast, Mtac2 deals with such errors in a more refined manner by throwing appropriately named exceptions. For example, the `VarAppearsInValue` exception signifies a leaked `nu` variable and `StuckTerm` indicates any kind of stuck term encountered by the interpreter, including when pattern matching on a variable or any other kind of opaque term. On the one hand, this means that the operational semantics of Mtac2 includes a large set of new exception-throwing rules corresponding to the various cases where Mtac1 terms could get blocked. On the other hand, raising an exception means that these errors can be caught and handled within a tactic using ordinary Mtac exception handling.

13.1 PRIMITIVES AND THEIR TYPES

Before we dive into the details of specific primitives, we first provide an overview of the primitives and their respective types in Figure 13.1. This list will be useful when discussing the reduction rules. Curly braces serve to indicate which arguments (typically, types) are implicit. We will omit them in the rules unless they play a part, in which case we use the `@` notation to instantiate them explicitly. We also omit type annotations on those variables that are of type `Type`.

We must note that this is not the complete set of primitives, but the ones that are considered stable (i.e., whose behavior will not be subject

to change in the foreseeable future) and that pose some interest from the perspective of the semantics (*e.g.*, we are not showing the primitives for querying or printing a term). We refer the reader to the code for the full set of primitives.

Universes

All Mtac2 primitives are universe polymorphic and come with a set of universe binders, and, optionally, universe constraints. These binders and constraints are affixed to the primitive's name using the `@{U1 .. | C1 ..}` syntax. Universe polymorphic types and type families used in the primitives' types—such as `Type`, Mtac2's monad `M`, `list`, or `option`—are also annotated with universes.

Figure 13.1: Mtac2 primitives with their respective types.

```

1 ret@{UA}:
2   forall {A:Type@{UA}}, A -> M@{UA} A
3 bind@{UA UB}:
4   forall {A:Type@{UA}} {B:Type@{UB}},
5     M@{UA} A -> (A -> M@{UB} B) -> M@{UB} B
6 mtry@{UA}:
7   forall {A:Type@{UA}},
8     M@{UA} A -> (Exception -> M@{UA} A) -> M@{UA} A
9 raise@{UA}:
10  forall {A:Type@{UA}}, Exception -> M@{UA} A
11 mfix1@{UA UP}:
12  forall {A:Type@{UA}} {P: A->Type@{UP}},
13    ((forall x: A, M@{UP} (P x)) -> (forall x: A, M@{UP} (P x))) ->
14    forall x: A, M@{UP} (P x)
15 mfix2@{UA1 UA2 UP}:
16  forall {A1:Type@{UA1}} {A2: A1->Type@{UA2}}
17    {P: forall (a1 : A1), A2 a1->Type@{UP}},
18    ((forall (x1: A1) (x2: A2 x1), M@{UP} (P x1 x2)) ->
19     (forall (x1: A1) (x2: A2 x1), M@{UP} (P x1 x2))) ->
20    forall (x1: A1) (x2: A2 x1), M@{UP} (P x1 x2)
21 gen_evar@{UA}:
22  forall (A:Type@{UA}), M@{UA} A
23 unify_cnt@{UA UP}:
24  forall {A:Type@{UA}} {P: A->Type@{UP}} (u: Unification) (x y: A),
25    M@{UP} (P y) -> M@{UP} (P x) -> M@{UP} (P x)
26 unify_univ@{UA UB}:
27  forall (A:Type@{UA}) (B:Type@{UB}) (u: Unification),
28    M@{UB} (option@{UB} (A->B))
29 nu@{UA UB}:
30  forall {A:Type@{UA}} {B:Type@{UB}} (n: name)
31    (body : option@{UA} A),
32    (A -> M@{UB} B) -> M@{UB} B
33 remove@{UA UB}:
34  forall {A:Type@{UA}} {B:Type@{UB}},
35    A -> M@{UA} B -> M@{UB} B
36 replace@{UA UB UC}:
37  forall {A:Type@{UA}} {B:Type@{UB}} {C:Type@{UC}} (x:A),
38    A = B -> M@{UC} C -> M@{UC} C
39 abs_fun@{UA UP UM | UM >= max(UA,UP)}:
40  forall {A:Type@{UA}} {P: A->Type@{UP}} (x: A),
41    P x -> M@{UM} (forall x, P x)
42 abs_let@{UA UP}:
43  forall {A:Type@{UA}} {P: A->Type@{UP}} (x: A) (y: A),
44    P x -> M@{UP} (let x := y in P x)
45 abs_prod_type@{UA UP UT UM | UT >= max(UA,UP), UM > UT}:
46  forall {A:Type@{UA}} (x : A) (P : Type@{UP}),
47    M@{UM} Type@{UT}
48 abs_prod_prop@{UA}:
49  forall {A:Type@{UA}} (x : A), Prop -> M@{Set} Prop
50 hyps@{UH UL}:
51  M (list@{UL} Hyp@{UH})

```

13.2 INTRODUCING MTAC2'S VIRTUAL MACHINE

Figure 13.2: Syntax of the virtual machine.

$$\begin{aligned}
V &::= \text{Code } t \mid \text{Ret } e \mid \text{Raise } e \\
C &::= \text{Bind } \cdot \cdot f \\
&\quad \mid \text{Nu } (x : A \text{ } [:= e]) \cdot \\
&\quad \mid \text{Remove } (x : A) \cdot \\
&\quad \mid \text{Replace } (x : A) \cdot \\
&\quad \mid \text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot \cdot f \\
Cs &::= C :: Cs \mid (\Omega, \Sigma, \Gamma, \cdot) \\
M &::= V :: Cs \\
\Sigma, \Sigma_\Delta &::= \emptyset \mid ?x : (\Gamma \vdash A \text{ } [:= e]), \Sigma_\Delta \\
\Gamma &::= [] \mid \Gamma, x : A \text{ } [:= e] \\
\Upsilon &::= v \cup \Upsilon \mid \emptyset \\
\Xi &::= v_0 \leq v_1 \cup \Xi \mid v_0 < v_1 \cup \Xi \mid \emptyset \\
\Omega, \Omega_\Delta &::= (\Upsilon, \Xi)
\end{aligned}$$

The virtual machine of Mtac2, whose syntax is presented in Figure 13.2, works as follows: it starts from an initial configuration, namely a *local* context Γ , a *meta*-context Σ , a universe context Ω , and the Mtac2 code to execute t :

$$\text{Code } t :: (\Omega, \Sigma, \Gamma, \cdot)$$

Then it performs the computation steps by pushing and popping Values and Continuations. Values appear only at the head of the stack, and they are the `Return` of a Rocq term, the `Raise` of an exception, or the execution of some piece of Mtac2's `Code`. The list of continuations consists of a series of Continuations ended by the initial configuration $(\Omega, \Sigma, \Gamma, \cdot)$. The dot $(\cdot)$ in the initial configuration and the continuations is a visual marking to indicate that it is expecting a value or another configuration to the left of the stack.

An important aspect of the continuations is that they include information required to continue the computation. For instance, the continuation `Bind` includes the functional argument f of the `bind` operator, and `Nu` includes the details of the new variable introduced by the `nu` operator. (Notation $[:= e]$ means that variable x or meta-variable $?x$ optionally have a definition.)

There is a special continuation for catching exceptions, `Mtry`, which includes contexts Σ_Δ and Ω_Δ , containing the newly introduced or instantiated meta-variables and universes with constraints, respectively, for the current session of the `mtry` operator.

Both local contexts and meta-contexts contain variable declarations

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

(standing for parameters and uninstantiated unification variables, respectively) *and* definitions (let-bound variables and instantiated unification variables, respectively). A meta-variable’s type is a *contextual type* (see Section 4.2 of [Zil+15]), which is a Rocq type A whose free variables are restricted to the ones declared in the local context Γ , noted $(\Gamma \vdash A)$. If the variable is defined with term t , its free variables should also be within those of Γ .

A universe context contains a set of universe variables (Υ) and a set of constraints between these variables (Ξ). Each universe variable is denoted with the letter v .

A step of computation, then, takes a Machine (a value and a stack of continuations) and produces a new one:

$$V :: Cs \xrightarrow{prog} V' :: Cs'$$

13.3 TYPING THE VIRTUAL MACHINE

The typing rules for typing a virtual machine are listed in Figure 13.3. In essence, they characterize the type of the resulting value of the computation.

To type a machine (**Machine**) we compute the universe, meta-, and local contexts from the list of continuations Cs in the following way:

$$\begin{aligned} \Omega(\Omega, \Sigma, \Gamma, \cdot) &= \Omega \\ \Omega(\text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f :: Cs) &= \Omega_\Delta \oplus \Omega Cs \\ \Omega(C :: Cs) &= \Omega Cs \\ \Sigma(\Omega, \Sigma, \Gamma, \cdot) &= \Sigma \\ \Sigma(\text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f :: Cs) &= \Sigma_\Delta \oplus \Sigma Cs \\ \Sigma(C :: Cs) &= \Sigma Cs \\ \Gamma(\Omega, \Sigma, \Gamma, \cdot) &= \Gamma \\ \Gamma(\text{Nu } (x : A := e)) \cdot :: Cs &= \Gamma Cs, (x : A := e) \\ \Gamma(\text{Remove } (x : A) \cdot :: Cs) &= \Gamma_1, \Gamma_2 \quad \text{where } \Gamma_1, x : A := e, \Gamma_2 = \Gamma Cs \\ \Gamma(\text{Replace } (x : B) \cdot :: Cs) &= \Gamma_1, (x : B := e), \Gamma_2 \\ &\quad \text{where } \Gamma_1, x : A := e, \Gamma_2 = \Gamma Cs \\ \Gamma(C :: Cs) &= \Gamma Cs \end{aligned}$$

We extend these functions to machines by projecting out the continuation, e.g. $\Omega(V :: C) = \Omega(C)$. We also often place a list of continuations Cs in the place where contexts are expected in order to mean $\Omega Cs; \Sigma Cs; \Gamma Cs$.

In essence, to obtain the resulting universe and meta-context from a list of continuations we just *merge in* each universe and meta-context stored in the **Mtry** continuations, and to obtain the resulting local context we add, remove, or replace each variable according to the corresponding

Machine Typing

$M : A$

Figure 13.3: Typing of the Virtual Machine.

$$\frac{\text{Machine} \quad \Omega \text{Cs}; \Sigma \text{Cs}; \Gamma \text{Cs} \vdash_v V : A \quad \text{Cs} : A \Rightarrow B}{V :: \text{Cs} : B}$$

Value Typing

$\Omega; \Sigma; \Gamma \vdash_v v : A$

$$\frac{\text{VCode} \quad \Omega; \Sigma; \Gamma \vdash t : \mathbb{M} A}{\Omega; \Sigma; \Gamma \vdash_v \text{Code } t : A} \quad \frac{\text{VRet} \quad \Omega; \Sigma; \Gamma \vdash e : A}{\Omega; \Sigma; \Gamma \vdash_v \text{Ret } e : A}$$

$$\frac{\text{VFail} \quad \Omega; \Sigma; \Gamma \vdash e : \text{Exception} \quad \Omega; \Sigma; \Gamma \vdash A : s}{\Omega; \Sigma; \Gamma \vdash_v \text{Fail } e : A}$$

Frame Typing

$\text{Cs} : A \Rightarrow B$

$$\frac{\text{CsInit} \quad \Omega; \Sigma; \Gamma \vdash A : s \quad \Omega; \Sigma; \Gamma \vdash A \equiv B}{(\Omega, \Sigma, \Gamma, \cdot) : A \Rightarrow B}$$

$$\frac{\text{CsCons} \quad \Omega \text{Cs}; \Sigma \text{Cs}; \Gamma \text{Cs} \vdash_c C : A \Rightarrow B \quad \text{Cs} : B \Rightarrow C}{C :: \text{Cs} : A \Rightarrow C}$$

Continuation Typing

$\Omega; \Sigma; \Gamma \vdash_c C : A_0 \Rightarrow A_1$

$$\frac{\text{CBind} \quad \Omega; \Sigma; \Gamma \vdash f : A \rightarrow \mathbb{M} B}{\Omega; \Sigma; \Gamma \vdash_c \text{Bind } \cdot f : A \Rightarrow B}$$

$$\frac{\text{CNu} \quad x \notin \text{dom}(\Gamma) \quad \Omega; \Sigma; \Gamma \vdash A : s \quad [\Omega; \Sigma; \Gamma \vdash e : A] \quad \Omega; \Sigma; \Gamma \vdash B : s'}{\Omega; \Sigma; \Gamma \vdash_c \text{Nu } (x : A [:= e]) \cdot B \Rightarrow B}$$

$$\frac{\text{CRemove} \quad x \notin \Gamma_2 \quad \Omega; \Sigma; \Gamma_1, \Gamma_2 \vdash B : s}{\Omega; \Sigma; \Gamma_1, x : A [:= e], \Gamma_2 \vdash_c \text{Remove } (x : A) \cdot B \Rightarrow B}$$

$$\frac{\text{CReplace} \quad \Omega; \Sigma; \Gamma \vdash B : s \quad \Omega; \Sigma; \Gamma \vdash A \equiv B \quad \Omega; \Sigma; \Gamma \vdash C : s'}{\Omega; \Sigma; \underbrace{\Gamma_1, x : A [:= e], \Gamma_2}_{\Gamma} \vdash_c \text{Replace } (x : B) \cdot C \Rightarrow C}$$

FMtry

$$\frac{\vdash \Omega_\Delta \oplus \Omega \quad \Omega_\Delta \oplus \Omega \vdash \Sigma_\Delta \oplus \Sigma \quad \Omega; \Sigma; \Gamma \vdash f : \text{Exception} \rightarrow \mathbb{M} A}{\Omega; \Sigma; \Gamma \vdash_c \text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f : A \Rightarrow A}$$

continuations. The merge operator $\oplus$ is defined as follows:

$$\begin{aligned}
\emptyset \oplus \Sigma &= \Sigma \\
\Sigma_{\Delta}, ?x : A [:= e] \oplus \Sigma &= \Sigma_{\Delta} \oplus \Sigma, ?x : A [:= e] \quad \text{if } ?x \notin \Sigma \\
\Sigma_{\Delta}, ?x : A := e \oplus \Sigma, ?x : A &= \Sigma_{\Delta} \oplus \Sigma, ?x : A := e \\
\emptyset \oplus \Upsilon &= \Upsilon \\
v \cup \Upsilon_{\Delta} \oplus \Upsilon &= \Upsilon_{\Delta} \oplus v \cup \Upsilon \\
\emptyset \oplus \Xi &= \Xi \\
v_0 \leq v_1 \cup \Xi_{\Delta} \oplus \Xi &= \Xi_{\Delta} \oplus v_0 \leq v_1 \cup \Xi \\
&\quad \text{if } (\Upsilon_{\Delta}, \Xi_{\Delta}) \oplus (\Upsilon, \Xi) = (\Upsilon_{\Delta} \oplus \Upsilon, \Xi_{\Delta} \oplus \Xi) \\
v_0 < v_1 \cup \Xi_{\Delta} \oplus \Xi &= \Xi_{\Delta} \oplus v_0 < v_1 \cup \Xi \\
&\quad \text{if } (\Upsilon_{\Delta}, \Xi_{\Delta}) \oplus (\Upsilon, \Xi) = (\Upsilon_{\Delta} \oplus \Upsilon, \Xi_{\Delta} \oplus \Xi)
\end{aligned}$$

What is implicit in this formulation is that there cannot be two instantiations for the same meta-variable.

By updating the contexts to the changes stacked by the different operators, we ensure the value V is typed in the same contexts it was produced. After checking that V has type A (that is, it is `Mtac2` code expected to produce an element of type A), we typecheck the continuations to confirm they expect an element of type A and produces an element of type B (noted $A \Rightarrow B$). Then, the machine will produce an element of type B .

The typing rules for the values are simple: for `Code` the term t must have type $\mathbb{M} \ A$, where A is the type of the rule, for `Ret` the term e must have type A , and for `Fail`, e must have the type of an exception. An interesting point to consider is that we expect terms to be *weakly-typed*, with the computed contexts *weakly well-formed*, and we ensure that this invariant is maintained by each reduction step. The exact definitions of weak typing of a term and weak well-formedness of contexts are presented in Sections 1.2.3 and 4.2 of [Zil15], but in essence, they establish that terms follow the typing rules of CIC (as in Rocq’s reference manual), *ignoring the termination checker*. We cannot ensure regular typing since `Mtac` depends on unification, which only ensure weak typing. See Section 15 of [ZS17] for further details.

To provide the type of a list of continuations we have two cases: the **CsInit** case is an identity, with the only restriction is that the type must be well-typed in the initial contexts (implicitly, these has to be well-formed too). The **CsCons** case composes the types from the continuation C with the recursive call with the list of continuations Cs . Note in this case how the intermediate type B must be well-formed in the contexts computed from Cs , that is, B cannot depend on any information from C .

To type the continuations we need to take care the specificities of each one: For `Nu`, we require that the variable is not present in the local context, that the type of the variable has type of a *sort* or kind s , and that the returning type B is also a sort in the original context (that is, without

[Zil15] Ziliani, “Interactive Typed Tactic Programming for the Coq Proof Assistant”

[ZS17] Ziliani et al., “A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading”

the variable being added). For `Remove` we must ensure that the returning type typechecks even when the variable is removed from the context, which also implies that the resulting context must be well-formed. We do not need to check that A (and optionally e) are well-typed, since they are expected to be the same as the one in the local context (whose well-formedness condition is assured). The `Replace` continuation is a bit more involved: here, we do check that the new type B given to x is a type, and is convertible to the original type A of x , written $\Omega; \Sigma; \Gamma \vdash A \equiv B$. As with the previous cases, the return type should have type sort in the original context. For `Mtry` we need to make sure the continuation f typechecks in the original Σ and Ω , and that their extension with the contexts Σ_Δ and Ω_Δ is possible.

13.4 SEMANTICS OF THE VIRTUAL MACHINE

In the coming sections we describe the execution of the different primitives, grouped accordingly to their purpose.

13.4.1 Monadic bind and fixed-points

Figure 13.4: Semantics of `bind` and `mfix`.

Ret
 $\text{Code } (\text{ret } e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } e :: \text{Cs}$

Raise
 $\text{Code } (\text{raise } e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise } e :: \text{Cs}$

Bind
 $\text{Code } (\text{bind } t \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } t :: \text{Bind} \cdot f :: \text{Cs}$

BindR
 $\text{Ret } e :: \text{Bind} \cdot f :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ e) :: \text{Cs}$

BindE
 $\text{Raise } e :: \text{Bind} \cdot f :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise } e :: \text{Cs}$

Mfix
 $\text{Code } (\text{mfix}_n \ f \ e_1 \ \dots \ e_n) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ (\text{mfix}_n \ f) \ e_1 \ \dots \ e_n) :: \text{Cs}$

Figure 13.4 shows the execution of `ret`, `raise`, `bind` and the `mfixn` constructs, with n an arbitrary natural number starting from 1, accounting for the number of arguments the fixed-point takes.¹

When binding the computation t with the continuation f (**Bind**), we put t as the computation to perform next, and add a frame $\text{Bind} \cdot f$ to the list of continuations Cs . Computation will continue then on t until it reaches a `ret` (**BindR**) or `raise` (**BindE**). In the first case, computation

¹ At the moment, for practical reasons, we only support n ranging from 1 to 5 in the `Mtac2` implementation. In Figure 13.1 we only showed the types for `mfix1` and `mfix2`, since the remaining three can be extrapolated easily from these.

continues with f applied to e , the returned value of t , and in the second case the exception is re-raised.

13.4.2 Term reduction

Figure 13.5: Semantics of term reduction.

$$\begin{array}{c}
 \text{ReduceLet} \\
 \hline
 \text{Cs} \vdash e \xrightarrow{\bar{r}} e'' \\
 \hline
 \text{Code } (\text{let } x := \text{reduce } r \text{ e in } e') :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } e' \{e''/x\} :: \text{Cs} \\
 \\
 \text{ReduceLetE} \\
 \hline
 e \text{ is not a reduction strategy} \\
 \hline
 \text{Code } (\text{let } x := \text{reduce } e \text{ e' in } e'') :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise RedFailure} :: \text{Cs} \\
 \\
 \text{Reduce} \\
 \hline
 \begin{array}{c}
 e \text{ is not a let-in with reduce} \quad \text{Cs} \vdash e \xrightarrow{w\beta\delta\iota\zeta}_+ e' \\
 \hline
 \text{Code } e :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } e' :: \text{Cs}
 \end{array} \\
 \\
 \text{ReduceE} \\
 \hline
 \begin{array}{c}
 \nexists e'. \text{Cs} \vdash e \xrightarrow{w\beta\delta\iota\zeta}_+ e' \quad e \text{ is not an application of an Mtac2 primitive} \\
 \hline
 e :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise StuckTerm} :: \text{Cs}
 \end{array}
 \end{array}$$

In Mtac we have two types of reduction rules: the rules for reducing applications of Mtac primitives, which we have been discussing so far (and will continue to discuss in subsequent sections), and the rules for allowing Mtac2 developers to reduce CIC terms implemented by Mtac2's `let-reduce` syntax. Here we discuss the latter, which we will refer to as *term reduction* rules.

Term reduction happens in two situations: when the user asks the interpreter to reduce a term explicitly by invoking the `reduce` primitive of Mtac2, or when the interpreter needs to reduce the term in evaluation position in order to reach an Mtac primitive. In both cases, terms are reduced according to the reduction rules of CIC. This reduction is written $\Omega; \Sigma; \Gamma \vdash e \xrightarrow{\rho} e'$ where the parameter ρ specifies what reduction is performed. We write $\Omega; \Sigma; \Gamma \vdash e \xrightarrow{\rho}_+ e'$ to indicate that the reduction must make progress, i.e. perform at least one step of reduction. There are different reduction strategies (e.g., *weak head*, *call-by-value*, etc.), as well as different types of reduction: standard β -reduction, ι -reduction (encompassing `match`-reduction and `fix`-reduction), ζ -reduction (reduction of `let-ins`), and δ -reduction (unfolding of defined constants or variables). When invoking the explicit `reduce` primitive, the user has the ability to specify which form of reduction they wish to use. Otherwise, the interpreter just performs full weak head reduction. Figure 13.5 shows the rules for these two situations. In these (and future) rules, we write $\bar{e}$ to denote the translation of a Rocq term e to its equivalent representation within the meta-theory.

The first two rules consider the first situation. Here, the user asks `Mtac2` to reduce a term explicitly by invoking `reduce r e`, which has the effect of reducing term `e` according to the reduction strategy specified by `r` of type `Reduction`. In `Mtac2` we have taken the original `Reduction` datatype of `Mtac1` and extended it to include all of the common strategies found in `Rocq` (Figure 13.6). As mentioned above, the API of `Rocq` is equipped with several reduction strategies: call-by-name (`hnf`), call-by-value (`cbv`), simplification (`simpl`, a heuristic to improve readability of terms with reducible patterns), and call-by-value executed by a virtual machine via `vm_compute` [GL02]. The first two, `hnf` and `cbv`, can be further parameterized over the kind of reduction (β , ι , ζ , or δ). In the case of δ -reduction, it is possible to specify which identifiers to unfold or which *not* to unfold.

[GL02] Grégoire et al., “A compiled implementation of strong reduction”

The `Rocq` types `Reduction` and `RedFlags` reify the various reduction strategies and reduction flags implemented by `Rocq`. We start with the constructors of `RedFlags`.

1. `RedBeta` represents β reduction.
2. `RedMatch` represents ι restricted to `match` reduction on inductive types.
3. `RedFix` represents ι restricted to fixpoint and co-fixpoint reduction.
4. `RedDelta` represents unrestricted δ reduction.
5. `RedDeltaOnly` represents δ reduction restricted to a given list of constants.
6. `RedDeltaExcept` represents δ reduction restricted to all constants *but* those specified.

The reduction strategies represented by `Reduction` are modeled after `Rocq`’s reduction strategies but attempt to be more principled than the reduction tactics exposed by `Ltac`, which often implement mixed strategies.

1. `RedNone` performs no reduction (useful when parametrizing a tactic over the reduction performed on its terms).
2. `RedSimpl` uses `Rocq`’s `simpl` tactic, which uses heuristics to *simplify* a terms.
3. `RedOneStep` performs a single head step that matches any of the given reduction flags.
4. `RedWhd` performs weak head reduction. Weak head reduction regards the term to be reduced as a (potentially unary) application and performs reduction according to the specified reduction rules *only* on prefixes of the application nodes. For example, given at least the `RedBeta` reduction flag, it reduces `(fun x => x) (cons (1 + 1) nil)` to `cons (1 + 1) nil` but no further.
5. `RedStrong` performs *strong reduction*, i.e., it performs *all* reductions allowed by the given reduction flags in any part of the given term.

6. `RedVmCompute` implements the virtual machine reduction strategy (`vm_compute`).

Note that `Mtac2`'s use of the `reduce` primitive marks a point of difference from `Mtac1`. In `Mtac1`, reduction was performed as part of the `ret` construct, which accepted an argument of the `Reduction` type in addition to the element to be returned. However, a problem with this approach was that a reduced term was not considered equal to the original term by the typechecker, leading to unnecessary coercions in the code. For example, in `Mtac1`'s code `x <- ret RedSimpl e; e'`, the definitional equality of `x` and `e` would not be known to `e'`, information that might be crucial to typecheck `e'`. In `Mtac2`, we solve this problem via the explicit `reduce` primitive, which employs an interesting trick: `reduce r` is defined simply to be the identity function, but when it appears as the head constant of the definition of a let-in, it signifies to the `Mtac2` interpreter that a reduction should be performed. Formally speaking, here is the definition of `reduce`:

Definition `reduce (r : Reduction) {A:Type} (x : A) := x.`

The example above, now in `Mtac2`, would be `let x := reduce RedSimpl e in e'`, which is equivalent, from the perspective of Rocq's typechecker, to `let x := e in e'`.

One might wonder if it is not a problem that this definition is *fragile* in the sense that `reduce` could get unintentionally unfolded. This situation, however, can only arise in three hypothetical scenarios: (1) if Rocq's typechecker unfolds it, (2) if the user unfolds it (e.g., by `reduce`-ing a `reduce` expression), or (3) if `Mtac2`'s interpreter incorrectly unfolds it. In the first case, it meant that Rocq's typechecker altered the term introduced by the user, a situation that to our understanding does (and certainly should) not happen. In the second case, if it is the user's intent to unfold a `reduce` expression, there is nothing we can do about it. In the third case, we know it cannot happen by inspecting the rules discussed below.

Coming back to the rules: In **ReduceLet**, `e` is reduced using the reduction strategy `r` (of type `Reduction`), so long as `r` is well-formed. If `r` is not well-formed (e.g., if `r` is a variable and not a constant of the `Reduction` type), we raise the `RedFailure` exception (**ReduceLetE**).

The last two rules, **Reduce** and **ReduceE**, consider the case when the `Mtac` interpreter performs reduction implicitly. In this case, we have two options: if the term is reducible using full weak head reduction, the term is reduced (**Reduce**); otherwise, if the term is not reducible and not an application of an `Mtac2` primitive, we raise the `StuckTerm` exception (**ReduceE**).

13.4.3 Meta-variable generation and instantiation

² The name is short for “generate (an) `ewar`” (in Rocq, meta-variables are called `existential variables`).

Meta-variable creation is performed by the operation `gen_ewar`² whose evaluation rule (**GenEvar**) is given in Figure 13.7. The existential variable is always created in the current local context Γ . We need to insert the new meta-variable in the last meta-context from the list of continuations Cs . We define a new operator $\oplus_F$ that, essentially, “commits” the changes

```

Inductive RedFlags : Set :=
| RedBeta | RedDelta | RedMatch
| RedFix | RedZeta
| RedDeltaC | RedDeltaX
| RedDeltaOnly: list dyn -> RedFlags
| RedDeltaBut: list dyn -> RedFlags.

Inductive Reduction : Set :=
| RedNone | RedSimpl
| RedOneStep: list RedFlags -> Reduction
| RedWhd: list RedFlags -> Reduction
| RedStrong: list RedFlags -> Reduction
| RedVmCompute.

```

Figure 13.6: Reduction API.

$$\begin{array}{c}
\text{GenEvar} \\
\hline
\Gamma = \Gamma Cs \quad Cs' = \{?x : (\Gamma \vdash A)\} \oplus_F Cs \\
\hline
\text{Code } (\text{gen_evar } A) :: Cs \xrightarrow{\text{prog}} \text{Ret } ?x[\hat{\Gamma}] :: Cs'
\end{array}$$

$$\begin{array}{c}
\text{UnifyCnt} \\
\hline
\Omega Cs; \Sigma Cs; \Gamma Cs \vdash e \approx_{\bar{u}} e' \triangleright \Omega_{\Delta}; \Sigma_{\Delta} \quad Cs' = (\Omega_{\Delta}, \Sigma_{\Delta}) \oplus_F Cs \\
\hline
\text{Code } (\text{unify_cnt } u e e' t t') :: Cs \xrightarrow{\text{prog}} \text{Code } t :: Cs'
\end{array}$$

$$\begin{array}{c}
\text{UnifyCntNot} \\
\hline
\Omega Cs; \Sigma Cs; \Gamma Cs \vdash e \not\approx_{\bar{u}} e' \\
\hline
\text{Code } (\text{unify_cnt } u e e' t t') :: Cs \xrightarrow{\text{prog}} \text{Code } t' :: Cs
\end{array}$$

$$\begin{array}{c}
\text{UnifyUniv} \\
\hline
\Omega Cs; \Sigma Cs; \Gamma Cs \vdash A \approx_{\bar{u} \leq} B \triangleright \Omega_{\Delta}; \Sigma_{\Delta} \quad Cs' = (\Omega_{\Delta}, \Sigma_{\Delta}) \oplus_F Cs \\
\hline
\text{Code } (\text{unify_univ } A B u) :: Cs \xrightarrow{\text{prog}} \text{Ret } (\text{Some } (\text{fun } x:A \Rightarrow x)) :: Cs'
\end{array}$$

$$\begin{array}{c}
\text{UnifyUnivNot} \\
\hline
\Omega Cs; \Sigma Cs; \Gamma Cs \vdash A \not\approx_{\bar{u} \leq} B \\
\hline
\text{Code } (\text{unify_univ } A B u) :: Cs \xrightarrow{\text{prog}} \text{Ret None} :: Cs
\end{array}$$

$$\begin{array}{c}
\text{UnifyCntIllFormed} \\
\hline
\bar{u} \text{ is not a unification strategy} \\
\hline
\text{Code } (\text{unify_cnt } u e e' t t') :: Cs \xrightarrow{\text{prog}} \text{Raise IllFormedUnif} :: Cs
\end{array}$$

$$\begin{array}{c}
\text{UnifyUnivIllFormed} \\
\hline
\bar{u} \text{ is not a unification strategy} \\
\hline
\text{Code } (\text{unify_univ } A B u) :: Cs \xrightarrow{\text{prog}} \text{Raise IllFormedUnif} :: Cs
\end{array}$$

Figure 13.7: Semantics of operators to create and instantiate meta-variables.

from the first meta-context to the second:

$$\begin{aligned}
 (\Omega_\Delta, \Sigma_\Delta) \oplus_F (\Sigma, \Gamma, \cdot) &= (\Omega_\Delta \oplus \Omega, \Sigma_\Delta \oplus \Sigma, \Gamma, \cdot) \\
 (\Omega_\Delta, \Sigma_\Delta) \oplus_F (\text{Mtry } (\Omega'_\Delta, \Sigma'_\Delta) \cdot f :: Cs) &= \\
 \text{Mtry } (\Omega_\Delta \oplus \Omega'_\Delta, \Sigma_\Delta \oplus \Sigma'_\Delta) \cdot f :: Cs &= \\
 (\Omega_\Delta, \Sigma_\Delta) \oplus_F (C :: Cs) &= C :: ((\Omega_\Delta, \Sigma_\Delta) \oplus_F Cs)
 \end{aligned}$$

Meta-variable instantiation is performed by unification. We have two operations: `unify_cnt` and `unify_univ`. `unify_cnt` unifies two terms (or types), and performs *conversion*. With conversion, if two terms e and e' are unifiable, then they are convertible, meaning that, if a term e'' has some type A , we can substitute e with e' in e'' and obtain a term whose type is equivalent to A . If terms e or e' have meta-variables, they will be instantiated in order to make the two terms convertible. In the case e is a meta-variable, then unification will instantiate it with whatever e' is, and therefore we consider this primitive a generalization of meta-variable instantiation.

The underlying meta-theoretic judgment is written $\Omega; \Sigma; \Gamma \vdash e \approx_u e' \triangleright \Omega_\Delta; \Sigma_\Delta$. It signifies that e and e' , valid terms in the initial context Σ and Ω , are convertible in the contexts $\Omega_\Delta \oplus \Omega$ and $\Sigma_\Delta \oplus \Sigma$ which result from unifying the terms e and e' w.r.t. the unification strategy u .³

These new contexts Ω_Δ and Σ_Δ reflect all the necessary universe constraints and instantiations of meta-variables, respectively. Concretely, the following facts hold.

Lemma 13.4.1 (Unification Validity). *If $\vdash \Omega; \Sigma; \Gamma$ and $\Omega; \Sigma; \Gamma \vdash e \approx_u e' \triangleright \Omega_\Delta; \Sigma_\Delta$ then $\vdash \Omega_\Delta \oplus \Omega; \Sigma_\Delta \oplus \Sigma; \Gamma$.*

Lemma 13.4.2 (Unification Convertibility). *If $\vdash \Omega; \Sigma; \Gamma$ and $\Omega; \Sigma; \Gamma \vdash e \approx_u e' \triangleright \Omega_\Delta; \Sigma_\Delta$ then $\Omega_\Delta \oplus \Omega; \Sigma_\Delta \oplus \Sigma; \Gamma \vdash e \equiv e'$.*

Given that then the two terms are convertible if unification succeeds, why are we not returning a proof of their equality ($e = e'$)? As a matter of fact, this was how the primitive worked in an earlier version of Mtac2. However, we decided that it was best to avoid constructing the equality proof, in order to simplify the interpreter and avoid unnecessary universes floating around (more on this last topic in Section 14.1). As it turns out, we can achieve the same via continuations. In order to see this clearly, we have to recall the type of `unify_cnt`:

$\forall \{A\} \{P : A \rightarrow \text{Type}\} (u : \text{Unification}) (e e' : A), M (P e') \rightarrow M (P e) \rightarrow M (P e)$

In this type, P denotes a multi-hole context in which we want to substitute e' for e when the unification of e and e' succeeds.

This idea is realized in `UnifyCnt`. When the two terms are unifiable, the execution continues with t having type $M (P e')$. But the primitive promises to return something of type $M (P e)$. The apparent mismatch is solved by observing that the two terms are convertible. In case the two terms are not unifiable, then the execution continues to t' (`UnifyCntNot`).

The second primitive for unification, `unify_univ`, is specific to types, and performs *cumulativity*: if two types are cumulatively unifiable, then

³ As a matter of fact, the actual implementation returns a new contexts Σ' and Ω' , and not just the additions. However, a promise of said primitive is that there exists Σ_Δ and Ω_Δ such that $\Sigma' = \Sigma_\Delta \oplus \Sigma$ and $\Omega' = \Omega_\Delta \oplus \Omega$.

the sort of the first one is smaller or equal—and not just equal as with conversion—than the sort of the second one. In particular, this means that there exists an injection from A to B or, to put it differently, that we can lift an element x of type A to be an element of type B . This is what **UnifyUniv** does: if the two types unify, then it returns a function from A to B , which is simply the identity function. In case they are not unifiable, then `None` is returned (**UnifyUnivNot**).

For `unify_univ` the underlying meta-theoretic judgment is written $\Omega; \Gamma; \Sigma \vdash e \approx_{u \leq} e' \triangleright \Omega_\Delta, \Sigma_\Delta$. It expresses that e and e' , valid terms in the initial universe and meta-contexts Ω and Σ , are cumulatively unifiable w.r.t. to the unification strategy u in the resulting universe and meta-contexts Ω_Δ and Σ_Δ in addition to the original ones. As with non-cumulative unification introduced above, these new contexts reflect all the necessary instantiations of meta-variables.

Before concluding this section, we quickly turn to the unification strategy parameter u . The parameter u has type `Unification`—a simple enumeration type. This type permits control on the unification algorithm used to equate the terms, and its values are the following:

- **UniCoq** It uses UniCoq, a plugin with a unifier formalized in [ZS17].
- **UniMatch** It uses UniCoq, but only permitting instantiations of meta-variables occurring on the first term, and only performing reduction on the second term. As the name suggests, this is the common unification strategy for pattern-matching: the first term is the pattern, and the second term is the scrutinee.
- **UniMatchNoRed** Similar to **UniMatch**, but also avoiding reducing the scrutinee.
- **UniEvarconv** It uses Rocq’s unification algorithm (called `evarconv` in the source files).

[ZS17] Ziliani et al., “A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading”

We also include rules (**UnifyCntIllFormed**, **UnifyUnivIllFormed**) that raise an exception in case this parameter is not well-formed. We refer the reader interested in understanding unification in detail to [ZS17].

13.4.4 Exception handling

Mtac2 offers the `mtry` primitive to handle exceptions. `mtry e f` behaves like e if e succeeds without an exception (**MtryR**). If e does raise an exception e' , `mtry` executes the handler f by passing it the exception e' . Thus, the execution continues with $f e'$ (**MtryE**). The most important aspect of `mtry`, though, is that it *backtracks* the meta-context in case an exception is raised. This means, in particular, that `mtry` can *undo* the introduction and instantiation of existential variables. This reset happens before executing the handler f , leaving no trace of the changes to the meta-context.

Let us assume that at the point of raising the exception e' the meta-context is Σ , but that the original meta-context (when `mtry` began) was

Figure 13.8: Semantics of `mtry`.

$$\begin{array}{c}
\text{Mtry} \\
\hline
\text{Code } (\text{mtry } e \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } e :: \text{Mtry } ((\emptyset, \emptyset), \emptyset) \cdot f :: \text{Cs} \\
\\
\text{MtryR} \\
\hline
\frac{\text{Cs}' = (\Omega_\Delta, \Sigma_\Delta) \oplus_F \text{Cs}}{\text{Ret } e :: \text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } e :: \text{Cs}'} \\
\\
\text{MtryE} \\
\hline
\frac{\text{Cs} \vdash e : \text{Exception}}{\text{Raise } e :: \text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ e) :: \text{Cs}} \\
\\
\text{MtryEENG} \\
\hline
\frac{\text{Cs} \not\vdash e : \text{Exception}}{\text{Raise } e :: \text{Mtry } (\Omega_\Delta, \Sigma_\Delta) \cdot f :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ \text{ExcNotGround}) :: \text{Cs}}
\end{array}$$

Σ' . Then we run into a potential issue: What if e' depends on variables from Σ that do not exist in Σ' ? A contrived but instructive example is the following Mtac2 code, in which the exception itself is an existential variable that is simply returned by the handler.

```
mtry (e <- evar Exception; raise e) (fun x => ret x)
```

It is clear that the exception e cannot be well-typed in the original meta-context, since that context does not contain e . Thus, if e were allowed to be returned by the handler, Mtac2's semantics would not be sound.

We solve this issue by replacing exceptions that are not well-typed in the previous context Σ' by the `ExcNotGround` exception, which—by virtue of having no arguments—is guaranteed to not depend on any meta-variable. This is implemented in `MtryEENG` and the corresponding precondition of `MtryE`.

A note about the local context.

Readers may wonder about potential leakage through the local context, which `mtry` does not reset. As is described in the next section, all primitives that manipulate the local context do so in a well-bracketed way—both for successful returns and exceptions. Thus, no changes to the local context can be leaked through exceptions.

13.4.5 Operations on the local context

When it comes to the local context Γ , we allow four kind of operations: introducing a variable (`nu`), removing a variable (`remove`), replacing the type of a variable (`replace`), and *weak*-reflecting the context (`hyps`).

Figure 13.9: Semantics of `nu`.

NuNone

$$\frac{x = \text{name_from}(\Gamma, \bar{n}) \quad x \notin \text{dom}(\Gamma)}{\text{Code } (\text{nu } n \ (\text{None } A) \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ x) :: \text{Nu } (x : A) \cdot :: \text{Cs}}$$

NuSome

$$\frac{x = \text{name_from}(\Gamma, \bar{n}) \quad x \notin \text{dom}(\Gamma)}{\text{Code } (\text{nu } n \ (\text{Some } A \ e) \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } (f \ x) :: \text{Nu } (x : A := e) \cdot :: \text{Cs}}$$

NuStuck

$$\frac{n \text{ not convertible to a name or } o \text{ not an option constructor}}{\text{Code } (\text{nu } n \ o \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise StuckTerm} :: \text{Cs}}$$

NuENEIC

$$\frac{x = \text{name_from}(\Gamma, \bar{n}) \quad x \in \text{dom}(\Gamma)}{\text{Code } (\text{nu } n \ o \ f) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise NameExistsInContext} :: \text{Cs}}$$

NuR

$$\frac{x \notin \text{FV}(e) \quad X \in \{\text{Ret}, \text{Raise}\}}{X \ e :: \text{Nu } (x : A := e') \cdot :: \text{Cs} \xrightarrow{\text{prog}} X \ e :: \text{Cs}}$$

NuVAIV

$$\frac{x \in \text{FV}(e) \quad X \in \{\text{Ret}, \text{Raise}\}}{X \ e :: \text{Nu } (x : A := e') \cdot :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise VarAppearsInValue} :: \text{Cs}}$$

Introducing variables.

The semantics of the `nu` operator is described in Figure 13.9. This operator takes a name n (described further below), optionally a definition d , and a function f . It executes f applied to x , a new variable added to the local context Γ . The variable x gets the name from n and is bound to d , if provided. Rules **NuNone** and **NuSome** begin the execution of the `nu` operator (without or with a let-defined variable, respectively) by using another run-time expression, `Nu`. In case the name is not well-formed or the variable exists in the context, an error is raised (respectively, **NuStuck** and **NuENEIC**).

When the execution of $f \ x$ reaches a value (**NuR** and **NuVAIV**), x is removed from the context by eliminating the `Nu` continuation. Note that the effects in the local context are all well-bracketed, so effectively the newer variable on the top of the context will be the variable introduced by the `nu` being executed. In the case a term e is returned or an exception is raised (**NuR**), we check that the variable x is not in the set of its free-variables, otherwise e would become ill-typed in the context without x . If x is free in e , then an exception is raised (**NuVAIV**). The corresponding exception, `VarAppearsInValue`, does not contain x and can thus safely escape the `nu` operator.

Figure 13.10: The inductive Rocq type `name`, representing different naming strategies for introducing new variables to the local context.

```
Inductive name :=
| FromStr (n: string)
| FreshFromStr (n: string)
| FreshFrom {A} (b: A)
| Generate.
```

Generating names.

The name for a variable is generated by the function `name_from( $\Gamma, n$ )`, which takes context Γ and an element n of the (Rocq) type `name`, which is shown in Figure 13.10. According to the constructors of `name`, it proceeds as follows: with `FromStr s` it takes the name as given by string s ; with `FreshFromStr s` it generates the fresh name from string s by either using s if it is already fresh or by appending the digit 0 to it, which is then incremented until the resulting string is fresh; with `FreshFrom e` it inspects the term e to find a base name, using the name of the binder in `forall` or `fun`, or using e itself if it is a named variable, and then proceeds as with `FreshFromStr`; and with `Generate` it generates a fresh name. Freshness is evaluated with respect to the local context Γ .

Figure 13.11: Semantics of `remove`.

Remove

$$\frac{\Gamma\text{Cs} = \Gamma_1, x : A[:= e'], \Gamma_2 \quad x \notin \text{FV}(B) \quad x \notin \text{FV}(e) \quad x \notin \text{FV}(\Gamma_2)}{\text{Code } (\text{remove } B \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } e :: \text{Remove } (x : A) \cdot :: \text{Cs}}$$

RemoveECRV

$$\frac{\Gamma\text{Cs} = \Gamma_1, x : A[:= e'], \Gamma_2 \quad x \in \text{FV}(B) \vee x \in \text{FV}(e) \vee x \in \text{FV}(\Gamma_2)}{\text{Code } (\text{remove } B \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Fail CannotRemoveVar} :: \text{Cs}}$$

RemoveENAV

$$\frac{e \text{ not a variable}}{\text{Code } (\text{remove } e \ e') :: \text{Cs} \xrightarrow{\text{prog}} \text{Fail NotAVariable} :: \text{Cs}}$$

RemoveV

$$\frac{X \in \{\text{Ret}, \text{Raise}\}}{X \ e :: \text{Remove } (x : A[:= e]) \cdot :: \text{Cs} \xrightarrow{\text{prog}} X \ e :: \text{Cs}}$$

Removing a variable from the context.

Removing a variable follows a similar pattern (Figure 13.11): the change to the context is recorded with a continuation, and it is removed prior to returning a value or raising an exception. Dual to `nu`, we need to check *first* that the variable can be safely removed (`Remove`). This amounts to checking that it occurs neither in the program e being executed, its type, nor in the context Γ_2 . If the argument is not a variable (`RemoveENAV`) or it cannot be safely removed (`RemoveECRV`), a proper exception is raised. When the evaluation reaches a value $X \ e$ (`RemoveV`), we just pop the

Remove continuation from the stack.

Figure 13.12: Semantics of `replace`.

Replace

$$\frac{p = \text{eq_refl} \quad x \in \text{dom}(\Gamma\text{Cs})}{\text{replace } A \ B \ C \ x \ p \ e :: \text{Cs} \xrightarrow{\text{prog}} \text{Code } e :: \text{Replace } (x : B) \cdot :: \text{Cs}}$$

ReplacePNER

$$\frac{p \neq \text{eq_refl}}{\text{replace } x \ p \ e :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise ProofNotEqRef1} :: \text{Cs}}$$

ReplaceENAV

$$\frac{e \text{ not a variable}}{\text{replace } e \ p \ e' :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise NotAVariable} :: \text{Cs}}$$

ReplaceV

$$\frac{X \in \{\text{Ret}, \text{Raise}\}}{X \ e :: \text{Replace } (x : A) \cdot :: \text{Cs} \xrightarrow{\text{prog}} X \ e :: \text{Cs}}$$

Replacing a hypothesis' type.

The combination of `remove` and `nu` is theoretically sufficient to substitute a variable's type (for instance, to apply reduction). However, any such implementation would have to `remove` the context Γ_2 to the right of x , replace x , and then re-introduce Γ_2 with `nu`. This, unfortunately, would be prohibitively slow. Thus, Mtac2 offers a primitive to replace the type of a hypothesis by a convertible type directly, without touching Γ_2 . To represent the condition that the new type is convertible, we ask for a proof of equality. More specifically, we ask for the proof to be `eq_refl`. This avoids potential issues with axioms and opaque equality proofs that can be used to create equality proofs that do not prove convertibility.

Shallow reflection of the context.

We make available the local context to the developer with the operation `hyps`. We say that this operation *shallowly* reflects the context because it only reflects the container structure (the list) in Rocq, without reflecting the types and terms it contains, as in other *reflective* tactic languages (see Section 15 for a comparison with other tactic languages). The rules for

this operation are:

$$\begin{array}{c}
 \text{Hyps-Ok} \\
 \hline
 \Omega; \Sigma; \Gamma \vdash \underline{\Gamma} : \text{list Hyp@}\{u\} \\
 \hline
 \text{hyps@}\{u\} :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } \underline{\Gamma} :: \text{Cs} \\
 \\
 \text{Hyps-Err} \\
 \hline
 \Omega; \Sigma; \Gamma \not\vdash \underline{\Gamma} : \text{list Hyp@}\{u\} \\
 \hline
 \text{hyps@}\{u\} :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise HypsUniverseException} :: \text{Cs}
 \end{array}$$

Where $\underline{\cdot}$ takes an object (in this case, the context) and shallowly reflects it to a corresponding Rocq term. In this case, we represent hypothesis by an inductive type, which is a container that consists of the variable itself and an optional body for let-bound variables. The inductive type, called `Hyp`, is given below.

```

Inductive Hyp@{U} : Prop :=
| ahyp {A : Type@{U}} : A -> moption@{U} A -> Hyp@{U}.

```

The `hyps` primitive is universe polymorphic in the universe level u used for its return value of type `list Hyp@{U}`. This level needs to be large enough to encompass the current goal's context. Since clients of `hyps` generally do not know statically what the current goal's context contains, the u is generally chosen to be another universe variable and clients need to take care to not restrict that variable too much. Since the `Mtac2` interpreter cannot rely on clients providing a large enough universe level, we guard the successful execution of the primitive by typechecking the reflected context against the return type of the `hyps` primitive. If the check fails, the primitive fails with a corresponding exception (see [Hyps-Err](#)).

Note that we can simply represent variables by themselves because values of `Hyp` are produced in the context they belong to. If that context changes, variables in `Hyp` values will need to be abstracted over just like variables in other terms.

A Note on `Prop`

Note that `ahyp` quantifies over `Type@{u}` and would thus put `Hyp` into a universe no less than $u+1$. While this is not a problem theoretically, the authors observed that it leads to universe issues in practice when writing `Mtac2` tactics. Given the complexity of the definitions involved together with the reliance on universe and universe constraint inference it is possible that the issues could be solved by manually annotating all universes. Unfortunately, even limited attempts at manual annotations have lead to a complete breakdown in maintainability and development productivity. We consider it infeasible to fully annotate all of `Mtac2`, even if only to find out whether the universe issues could be fixed in Rocq itself.

We instead choose to put `Hyp` into `Prop`. This drastically limits the ability to perform computation on values of type `Hyp` in pure Rocq. However, we can still write `Mtac2` programs that perform any desired computation so the loss of expressivity is minimal.

13.4.6 Abstracting a variable

The last set of primitives of interest are displayed in Figure 13.13. They all follow a similar pattern: given a variable x and a term e , they bind x with either (1) a lambda-abstraction (`abs_fun`), (2) a product (`abs_prod_type` and `abs_prod_prop`), or (3) a let-binding (`abs_let`). But these operations cannot be always performed: there are certain conditions that have to be met. For starters, it is necessary that e does not have any variable whose type depends on x . To see how this can be problematic, consider the context $\Gamma = [x : \text{Type}, y : x]$ and term `@Dyn x y` of type `dyn`. If we bind x in the term, we will end up with the ill-typed term `fun z : Type => @Dyn z y`. To prevent such situations, we intersect the free variables of e with the set γ of variables in the context that, directly or indirectly, depends on x (as expressed by the dependencies function). If the resulting set is larger than $\{x\}$, then we arrive at a situation like the one from the example above, and raise an exception (e.g., `AbsFunEDep`).

Mtac1 only included `abs_fun` (named `abs`), and its formulation differs a bit. On the one hand, Mtac2's dependency check is more permissive than the corresponding check in Mtac1, which simply disallowed any dependency on the variable x in the context Γ_2 . On the other hand, Mtac2 solves a soundness issue that was present in Mtac1's `abs_fun` primitive. To understand this soundness issue, consider the context $[x := 2]$ and the term `eq_refl` of type $(x = 2)$, which is well-typed because x and `2` are convertible in the current context. If we were allowed to abstract x with `abs_fun`, we would obtain an ill-typed term: `fun x => eq_refl : x = 2`. This abstraction was unsoundly allowed in Mtac1. In Mtac2, `abs_fun` does not allow let-bound variables to be abstracted over and instead introduces a new primitive, `abs_let` to handle let bindings. As a requirement, when performing an `abs_let x e e'`, the new definition for x should be convertible with the definition in the local context. But if x does not have a definition, then it is OK to perform the let-binding.

Universes of `abs_prod_type`

Recall the full signature of `abs_prod_type`:

```
abs_prod_type@{UA UP UT UM | UT >= max(UA,UP), UM > UT}:
  ∀{A : Type@{UA}} (x : A) (P : Type@{UP}) -> M@{UM} Type@{UT}
```

Since `abs_prod_type` generates a product in the `Type` sort, we have to ensure that the primitive's universes and universe constraints reflect the exact universe size of the generated term. As prescribed by the Prod-Type typing rule[The24], the universe of a product must be large enough to contain both domain and codomain of the product. Thus, in our primitive, we set the generated term's universe `UT` to be exactly the maximum of the domain's universe `UA` and the codomain's universe `UP`. (`UM` exists solely to specify the universe of the monad `M`.)

[The24] The Coq Development Team, “The Coq Reference Manual – Release 8.19.0”

Figure 13.13: Semantics of `abs_...` operators.

AbsFun	
$\frac{\Gamma\text{Cs} = \Gamma_1, x : A', \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \subseteq \{x\} \quad \gamma \cap \text{FV}(P) = \emptyset}{\text{Code } (\text{@abs_fun } A \ P \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } (\text{fun } y : A \Rightarrow e\{y/x\}) :: \text{Cs}}$	
AbsFunEDep	
$\frac{\Gamma\text{Cs} = \Gamma_1, x : A', \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \not\subseteq \{x\} \vee \gamma \cap \text{FV}(P) \neq \emptyset}{\text{Code } (\text{@abs_fun } A \ P \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise AbsDependencyError} :: \text{Cs}}$	
AbsEDef	
$\frac{\Gamma\text{Cs} = \Gamma_1, x := _, \Gamma_2 \quad X \in \{\text{abs_fun}, \text{abs_prod_type}, \text{abs_prod_prop}\}}{\text{Code } (X \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise AbsVariableIsADefinition} :: \text{Cs}}$	
AbsENAV	
$\frac{e \text{ not a variable} \quad X \in \{\text{abs_fun}, \text{abs_prod_type}, \text{abs_prod_prop}\}}{\text{Code } (X \ e \ e') :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise NotAVariable} :: \text{Cs}}$	
AbsProd	
$\frac{\Gamma\text{Cs} = \Gamma_1, x : A', \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \subseteq \{x\} \quad X \in \{\text{@abs_prod_type}, \text{@abs_prod_prop}\}}{\text{Code } (X \ A \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } (\forall y : A, e\{y/x\}) :: \text{Cs}}$	
AbsProdEDep	
$\frac{\Gamma\text{Cs} = \Gamma_1, x : A', \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \not\subseteq \{x\} \quad X \in \{\text{@abs_prod_type}, \text{@abs_prod_prop}\}}{\text{Code } (X \ A \ x \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise AbsDependencyError} :: \text{Cs}}$	
AbsLet	
$\frac{\Gamma\text{Cs} = \Gamma_1, x : A'[:= e''], \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \subseteq \{x\} \quad \gamma \cap \text{FV}(P) = \emptyset \quad [\text{Cs} \vdash e' \equiv e'']}{\text{@abs_let } A \ P \ x \ e' \ e :: \text{Cs} \xrightarrow{\text{prog}} \text{Ret } (\text{let } z : A \ := \ e' \ \text{in } e\{z/x\}) :: \text{Cs}}$	
AbsLetEDep	
$\frac{\Gamma = \Gamma_1, x : A'[:= e''], \Gamma_2 \quad \gamma = \text{dependencies}(\Gamma_2, x) \quad \gamma \cap \text{FV}(e) \not\subseteq \{x\} \vee \gamma \cap \text{FV}(P) \neq \emptyset \vee [\text{Cs} \vdash e' \not\equiv e'']}{\text{Code } (\text{@abs_let } A \ P \ x \ e' \ e) :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise AbsDependencyError} :: \text{Cs}}$	
AbsLetENAV	
$\frac{e \text{ not a variable}}{\text{Code } (\text{abs_let } e \ e' \ e'') :: \text{Cs} \xrightarrow{\text{prog}} \text{Raise NotAVariable} :: \text{Cs}}$	

13.5 THEORY

The proofs require some standard results for the underlying calculus:

Theorem II.1 (Strengthening). *Let $\Gamma = \Gamma_1, x : A[:= e']$, Γ_2 , Σ , Ω , e and B such that*

- $x \notin FV(e)$
- $x \notin FV(\Gamma_2)$
- $x \notin FV(B)$
- $\Omega; \Sigma; \Gamma \vdash e : B$

Then $\Omega; \Sigma; \Gamma_1, \Gamma_2 \vdash e : B$.

Theorem II.2 (Weakening). *Let $\Gamma = \Gamma_1, \Gamma_2$, Σ , Ω , e , B , A , x , e' and sort s such that*

- $\Omega; \Sigma; \Gamma_1, \Gamma_2 \vdash e : B$
- $x \notin FV(\Gamma_2)$
- $x \notin FV(e)$
- $x \notin FV(B)$
- $\Omega; \Sigma; \Gamma_1 \vdash A : s$
- $\Omega; \Sigma; \Gamma_1 \vdash x : A$

Then $\Omega; \Sigma; \Gamma_1, x : A[:= e'], \Gamma_2 \vdash e : B$.

Theorem II.3 (Convertibility). *Let Γ , Σ , Ω , e , e' and A such that*

$$\Omega; \Sigma; \Gamma \vdash e \equiv e'$$

Further, let sort s , type B and term f such that

- $\Omega; \Sigma; \Gamma, x : A \vdash B : \text{Sort}$,
- $\Omega; \Sigma; \Gamma, x : A \vdash f : B$,
- $\Omega; \Sigma; \Gamma \vdash f\{e/x\} : B\{e/x\}$,

Then,

$$\Omega; \Sigma; \Gamma \vdash f\{e'/x\} : B\{e'/x\}$$

Definition 13.5.1 (Initial Contexts). *We define projections from a machine to the contexts contained in its initial configuration.*

$$\Omega_i(\Omega, \Sigma, \Gamma, \cdot) = \Omega$$

$$\Omega_i(C :: Cs) = \Omega_i Cs$$

$$\Sigma_i(\Omega, \Sigma, \Gamma, \cdot) = \Sigma$$

$$\Sigma_i(C :: Cs) = \Sigma_i Cs$$

$$\Gamma_i(\Omega, \Sigma, \Gamma, \cdot) = \Gamma$$

$$\Gamma_i(C :: Cs) = \Gamma_i Cs$$

Lemma 13.5.1 (Frame Typing Result Well-Formedness). *If $Cs : A \Rightarrow B$ then there exists sort s s.t. $\Omega_i Cs; \Sigma_i Cs; \Gamma_i Cs \vdash B : s$.*

Proof. By induction on $Cs : A \Rightarrow B$, with the case of **CsInit** being trivial by convertibility, and that of **CsCons** following by induction. $\square$

Lemma 13.5.2 (Continuation Typing Well-Formedness). *If $Cs \vdash_c C : A \Rightarrow B$ then there exists sorts s, s' s.t. $(C :: Cs) \vdash A : s$ and $(C :: Cs) \vdash B : s'$.*

Proof. By case on the continuation:

- **CBind**: Implied by typing judgment on f .
- **CNu**: By weakening.
- **CReplace**, **CRemove**: Trivial.
- **FMtry**: By weakening the typing judgment on f .

$\square$

Lemma 13.5.3 (Frame Typing Input Well-Formedness). *If $Cs : A \Rightarrow B$ then there exists Sort s s.t. $Cs \vdash B : s$.*

Proof. By inversion on $Cs : A \Rightarrow B$. In the case of **CsInit** the statement holds by assumption. In the case of **CsCons**, the statement follows from 13.5.2. $\square$

Lemma 13.5.4 (Frame Typing Respects Right Convertibility). *If $Cs : A \Rightarrow B$, and $Cs \vdash B \equiv C$ then $Cs : A \Rightarrow C$.*

Proof. By induction on $Cs : A \Rightarrow B$. In the case of **CsInit** the statement holds by transitivity of $\equiv$. In the case of **CsCons** the statement holds by induction. $\square$

Theorem II.4 (Frame Typing Respects Left Convertibility). *If $\vdash \Omega Cs; \Sigma Cs; \Gamma Cs, Cs : B \Rightarrow A$, and $Cs \vdash B \equiv C$, then $Cs : C \Rightarrow A$.*

Proof. By induction on $Cs : B \Rightarrow A$.

- **CsInit**. We have $Cs = (\Omega_0, \Sigma_0, \Gamma_0, \cdot)$ for some $\Omega_0, \Sigma_0, \Gamma_0$, $\Omega_0; \Sigma_0; \Gamma_0 \vdash B : s$, $\Omega_0; \Sigma_0; \Gamma_0 \vdash B \equiv A$, and our initial assumption $\Omega_0; \Sigma_0; \Gamma_0 \vdash B \equiv C$. To show: (1) $\Omega_0; \Sigma_0; \Gamma_0 \vdash C : s$, and (2) $\Omega_0; \Sigma_0; \Gamma_0 \vdash C \equiv A$. Obligation (1) follows from $\Omega_0; \Sigma_0; \Gamma_0 \vdash B \equiv C$ and compatibility of convertibility and typing. Obligation (2) follows from $\Omega_0; \Sigma_0; \Gamma_0 \vdash B \equiv C$ and transitivity of convertibility.
- **CsCons**. We have $Cs = C :: Cs'$, a type D s.t. $Cs' \vdash_c C : B \Rightarrow D$, $Cs' : D \Rightarrow A$, and our initial assumption $Cs \vdash B \equiv E$.

To show: (1) $Cs \vdash_c C : E \Rightarrow D$. and (2) $Cs' : D \Rightarrow A$. (2) is our inductive hypothesis. For (1), we proceed by case distinction on C .

- $C = \text{Mtry } (\Omega'_\Delta, \Sigma'_\Delta) \cdot f$. We have $D = B$ and $Cs' \vdash f : \text{Exc} \rightarrow \mathbb{M} B$.
To show: (1) $\vdash \Omega'_\Delta \oplus \Omega Cs'$, (2) $\Omega'_\Delta \oplus \Omega Cs' \vdash \Sigma'_\Delta \oplus \Sigma Cs'$, (3) $Cs' \vdash f : \text{Exc} \rightarrow \mathbb{M} E$, and
(1) and (2) follow from $Cs' \vdash_c C : B \Rightarrow D$. (3) follows from $\Omega_0; \Sigma_0; \Gamma_0 \vdash B \equiv E$, and $Cs' \vdash f : \text{Exc} \rightarrow \mathbb{M} B$. (4) follows from our assumptions.

In all cases below: We have $\Omega Cs = \Omega Cs'$, $\Sigma Cs = \Sigma Cs'$, and $\Gamma Cs = \Gamma Cs'$, as well as $Cs = C :: (\Omega_\Delta, \Sigma_\Delta) \oplus_F Cs'$.

- $C = \text{Bind} \cdot f$.
We have $Cs \vdash f : A \rightarrow \mathbb{M} B$.
To show $Cs \vdash f : A \rightarrow \mathbb{M} E$. This follows from $Cs \vdash B \equiv E$ and compatibility of typing with convertibility.
- $C = \text{Nu } (x : E[:= e]) \cdot$.
We have $D = B$, $x \notin \text{dom}(\Gamma Cs)$, $Cs \vdash E : s$, $[Cs \vdash e : E]$, and $Cs \vdash B : s'$.
To show: $\Omega(Cs); \Sigma(Cs); \Gamma(Cs) \vdash E : s'$. This follows from $Cs \vdash B \equiv E$ and compatibility of typing with convertibility.
- $C = \text{Remove } (x : E[:= e]) \cdot$.
We have $D = B$, $\Gamma Cs = \Gamma_1, x : E[:= e], \Gamma_2$, $x \notin \Gamma_2$, and $\Omega; \Sigma; \Gamma_1, \Gamma_2 \vdash B : s$.
To show: $\Omega; \Sigma; \Gamma_1, \Gamma_2 \vdash E : s$. This follows from $Cs \vdash B \equiv E$ and compatibility of typing with convertibility.
- $C = \text{Replace } (x : F) \cdot$.
We have $D = B$, $\Gamma Cs = \Gamma_1, x : E[:= e], \Gamma_2$, $\Omega; \Sigma; \Gamma \vdash F : s$, $\Omega; \Sigma; \Gamma \vdash E \equiv F$, and $\Omega; \Sigma; \Gamma \vdash B : s'$.
To show: $\Omega; \Sigma; \Gamma \vdash E : s'$. This follows from $Cs \vdash B \equiv E$ and compatibility of typing with convertibility.

□

Lemma 13.5.5 (Frame Typing Monotonicity w.r.t. $\oplus_F$). *If $\vdash \Omega Cs; \Sigma Cs; \Gamma Cs \vdash \Omega_\Delta \oplus \Omega Cs; \Sigma_\Delta \oplus \Sigma Cs; \Gamma Cs, Cs : B \Rightarrow A$, then $(\Omega_\Delta, \Sigma_\Delta) \oplus_F Cs : B \Rightarrow A$.*

Proof. By induction on $Cs : B \Rightarrow A$. All obligations follow from the monotonicity of typing and convertibility w.r.t. $\oplus$. □

Lemma 13.5.6 (Well-formedness of contexts). *For all machines M and type A such that $M : A$, then $\vdash \Omega M; \Sigma M; \Gamma M$.*

Proof. By induction in the typing relation, noting that a Rocq term being well-typed implies that the contexts in which it is typed are well-formed. □

Theorem II.5 (Progress). *For all machines M and type A , if $M : A$ then one of the following holds:*

- $M = \text{Ret } e :: (\Omega, \Sigma, \Gamma, \cdot)$
- $M = \text{Fail } e :: (\Omega, \Sigma, \Gamma, \cdot)$

- There exists M' s.t. $M \xrightarrow{prog} M'$.

Proof. By inversion on $M : A$. Observe that all cases have associated steps and that all conditional steps come in sets with mutually-exclusive conditions that, when combined, are exhaustive. The only rules where there might be a missing case is the Abs family: the term e referring to the variable to abstract is either considered part of the local context (that is, it is a variable), or it is considered not to be a variable. One could argue that there could be a third case: the term being a variable but not in the context. That is ruled out by noting that then the machine M would not be well-typed, falsifying the hypothesis. $\square$

Theorem II.6 (Preservation). *Let M , M' , and A such that $M : A$ and $M \xrightarrow{prog} M'$.*

Then $M' : A$,

and there exists $\Omega_{i,\Delta}$, $\Sigma_{i,\Delta}$ such that $\Omega_i(M') = \Omega_i(M) \oplus \Omega_{i,\Delta}$, $\Sigma_i(M') = \Sigma_i(M) \oplus \Sigma_{i,\Delta}$, and $\Gamma_i(M') = \Gamma_i(M)$.

Proof. We proceed by case distinction on $M \xrightarrow{prog} M'$.

Bind: We have $\text{Code } (\text{bind } t \ f) :: Cs : A$ for some t and f . By inversion we know that $Cs \vdash_v \text{Code } (\text{bind } t \ f) : B$ and $Cs : B \Rightarrow A$ (for some B), and with another two inversions that $Cs \vdash t : \mathbb{M} \ C$ and $Cs \vdash f : C \rightarrow \mathbb{M} \ B$, for some C . Then,

$$\text{Machine} \frac{\text{VCode} \frac{Cs \vdash t : \mathbb{M} \ C}{Cs \vdash_v \text{Code } t : C} \quad \frac{\frac{Cs \vdash f : C \rightarrow \mathbb{M} \ B}{Cs \vdash_c \text{Bind} \cdot f : C \Rightarrow B} \text{CBind} \quad Cs : B \Rightarrow A}{\text{Bind} \cdot f :: Cs : C \Rightarrow A} \text{CsCons} \quad \frac{}{\underbrace{\text{Code } t :: \text{Bind} \cdot f :: Cs : A}_{M'}}$$

We have $\Omega_i(M') = \Omega_i(M)$, $\Sigma_i(M') = \Sigma_i(M)$, and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

GenEvar: We have $M' = \text{Ret } ?x[\hat{\Gamma}Cs] :: Cs'$ with $Cs' = \{?x : (\Gamma Cs \vdash B)\} \oplus_F Cs$.

$$\text{Machine} \frac{\text{VRet} \frac{\text{Hyp. with } \Gamma Cs = \Gamma Cs' \quad \frac{Cs' \vdash ?x[\hat{\Gamma}Cs] : B}{Cs' \vdash_v \text{Ret } ?x[\hat{\Gamma}Cs] : B}}{\text{Ret } ?x[\hat{\Gamma}Cs] :: Cs' : A} \quad \frac{\frac{\Omega Cs \vdash \{?x : (\Gamma Cs \vdash B)\} \oplus \Sigma Cs}{Cs : B \Rightarrow A} ?x \text{ fresh}}{Cs' : B \Rightarrow A} \text{13.5.5}$$

We have $\Omega_i(M') = \Omega_i(M)$ and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$. The choice of $\Sigma_{i,\Delta}$ depends on the shape of Cs' .

- Cs' contains $\mathbb{M}\text{try}$ nodes. We have $\Sigma_i(M') = \Sigma_i(M)$. We choose $\Sigma_{i,\Delta} = \emptyset$.
- Cs' does not contain $\mathbb{M}\text{try}$ nodes. We have $\Sigma_i(M') = ?x : (\Gamma Cs \vdash A) \oplus \Sigma_i(M)$. We choose $\Sigma_{i,\Delta} = ?x : (\Gamma Cs \vdash B)$.

Remove: We have $\text{Code } (\text{remove } B \ C \ x \ e) :: Cs : A$. By inversion we know that $Cs \vdash \text{remove } B \ C \ x \ e : \mathbb{M} \ C$, and, with another inversion, that $Cs \vdash e : \mathbb{M} \ C$. By assumption, there exists Γ_1 and Γ_2 such that $\Gamma Cs = \Gamma_1, x : B[:= e'], \Gamma_2$, and by *strengthening* (noting $x \notin \text{FV}(e)$ and $x \notin \text{FV}(\Gamma_2)$ and $x \notin \text{FV}(C)$), we get that $\Omega Cs; \Sigma Cs; \Gamma_1, \Gamma_2 \vdash e : \mathbb{M} \ C$. Note also that $\Omega Cs' = \Omega Cs$, $\Sigma Cs' = \Sigma Cs$, $\Gamma Cs' = \Gamma_1, \Gamma_2$, and thus $Cs' \vdash e : \mathbb{M} \ C$.

Therefore:

$$\begin{array}{c}
 \text{VCode} \quad \frac{Cs' \vdash e : \mathbb{M} \ C}{Cs' \vdash_v \text{Code } e : C} \quad \frac{\frac{\Omega Cs; \Sigma Cs; \Gamma_1, \Gamma_2 \vdash B : s}{Cs \vdash_c \text{Remove } (x : B) \cdot : C \Rightarrow C} \text{CRemove} \quad \frac{Cs : C \Rightarrow A}{Cs' : C \Rightarrow A} \text{CsCons}}{\frac{\text{Code } e :: \underbrace{\text{Remove } (x : B) \cdot}_{Cs'} :: Cs : A}{\underbrace{\hspace{10em}}_{M'}}} \text{Machine}
 \end{array}$$

We have $\Omega_i(M') = \Omega_i(M)$, $\Sigma_i(M') = \Sigma_i(M)$, and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

Replace: We have $\text{Code } (\text{replace } B \ C \ D \ x \ p \ e) :: Cs : A, p = \text{eq_refl}, x \in \text{dom}(\Gamma Cs)$. By inversion, we have $Cs \vdash B : s$, $Cs \vdash D : s'$, $Cs \vdash e : \mathbb{M} \ B$, as well as $Cs \vdash p : B = C$. With $p = \text{eq_refl}$ it follows that $Cs \vdash B \equiv C$. We also have $\Omega Cs' = \Omega Cs$, $\Sigma Cs' = \Sigma Cs$, and, thus—by compatibility of typing with convertibility of variable types— $Cs' \vdash e : \mathbb{M} \ C$.

Therefore:

$$\begin{array}{c}
 \text{VCode} \quad \frac{Cs' \vdash e : \mathbb{M} \ D}{Cs' \vdash_v \text{Code } e : D} \quad \frac{\frac{Cs' \vdash C : s \quad Cs' \vdash B \equiv C \quad Cs' \vdash D : s'}{Cs \vdash_c \text{Replace } (x : C) \cdot : D \Rightarrow D} \text{CReplace} \quad \frac{Cs : D \Rightarrow A}{Cs' : D \Rightarrow A} \text{CsCons}}{\frac{\text{Code } e :: \underbrace{\text{Replace } (x : D) \cdot}_{Cs'} :: Cs : A}{\underbrace{\hspace{10em}}_{M'}}} \text{Machine}
 \end{array}$$

We have $\Omega_i(M') = \Omega_i(M)$, $\Sigma_i(M') = \Sigma_i(M)$, and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

UnifyCnt We have $M = \text{Code } (\text{unify_cnt } u \ e \ e' \ t \ t') :: Cs, Cs : B \Rightarrow A$, a predicate P s.t. $B = P(e)$ and $Cs \vdash \text{Code } (\text{unify_cnt } u \ e \ e' \ t \ t') : \mathbb{M} \ (P \ e)$, $\Omega Cs; \Sigma Cs; \Gamma Cs \vdash e \approx_{\bar{u}} e' \triangleright \Omega_\Delta; \Sigma_\Delta$, $Cs' = (\Omega_\Delta, \Sigma_\Delta) \oplus_F Cs$, and $M' = \text{Code } t :: Cs'$.

To show: (1) $Cs' \vdash \text{Code } t : \mathbb{M} \ (P \ e)$ and (2) $Cs : B \Rightarrow A$.

By inversion of $Cs \vdash \text{Code } (\text{unify_cnt } u \ e \ e' \ t \ t') : \mathbb{M} \ (P \ e)$ we have $Cs \vdash t : \mathbb{M} \ (P \ e')$. From this obligation (1) follows with 13.4.2 and

compatibility of typing with convertibility. Obligation (2) follows from 13.4.1 and 13.5.5.

We have $\Omega_i(M') = \Omega_i(M)$, $\Sigma_i(M') = \Sigma_i(M)$, and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

AbsFun We have $V = @abs_fun\ B\ P\ x\ e$, $Cs \vdash_v @abs_fun\ B\ P\ x\ e : (\text{forall } x : B, P\ x)$, $Cs : (\text{forall } x : B, P\ x) \Rightarrow A$, $\Gamma Cs = \Gamma_1, x : B, \Gamma_2$, $\gamma = \text{dependencies}(\Gamma_2, x)$, $\gamma \cap FV(e) \subseteq \{x\}$, $\gamma \cap FV(P) = \emptyset$, and $M' = \text{Ret } (\text{fun } y : B \Rightarrow e\{y/x\}) :: Cs$.

To show: $Cs \vdash_v \text{Ret } (\text{fun } y : B \Rightarrow e\{y/x\}) : (\text{forall } x : B, P\ x)$, i.e. $Cs \vdash (\text{fun } y : B \Rightarrow e\{y/x\}) : (\text{forall } x : B, P\ x)$. This follows by inversion.

We have $\Omega_i(M') = \Omega_i(M)$, $\Sigma_i(M') = \Sigma_i(M)$, and $\Gamma_i(M') = \Gamma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

Universes

Recall that the type of `abs_fun` (with some alpha-renaming) is as follows:

```
abs_fun@{UB UE UM | UM >= max(UB, UE)}:
  forall {B:Type@{UB}} {e: B->Type@{UE}} (x: A),
    P x -> M@{UM} (forall (x : A), P x)
```

We have to show that $Cs \vdash ((\text{fun } y : B \Rightarrow e\{y/x\})) : \forall x, P\ x$ which, as far as universes are concerned, amounts to showing that there exists a sort s s.t. $Cs \vdash \forall x, P\ x : s$. We pick $s = \text{Type@}\{UM\}$. It remains to show that (1) $Cs \vdash B : \text{Type@}\{UM\}$, and (2) $(\Omega Cs, \Sigma Cs, (\Gamma Cs, y : B)) \vdash e\{y/x\} : \text{Type@}\{UM\}$. By assumption, we have $Cs \vdash B : \text{Type@}\{UB\}$, $\Omega Cs, \Sigma Cs, (\Gamma Cs, y : B) \vdash e\{y/x\} : \text{Type@}\{UE\}$, and $UM \geq \max(UB, UE)$. (1) and (2) follow by cumulativity.

AbsProd We demonstrate the reasoning only for `abs_prod_type`.

We have $V = @abs_prod_type\ B\ x\ e$, $Cs \vdash_v @abs_prod_type\ B\ x\ e : \text{Type}$, $Cs : \text{Type} \Rightarrow A$, $\Gamma Cs = \Gamma_1, x : B, \Gamma_2$, $\gamma = \text{dependencies}(\Gamma_2, x)$, $\gamma \cap FV(e) \subseteq \{x\}$, $\gamma \cap FV(P) = \emptyset$, and $M' = \text{Ret } (\text{forall } y : B \Rightarrow e\{y/x\}) :: Cs$.

To show: $Cs \vdash_v \text{Ret } (\text{forall } y : B \Rightarrow e\{y/x\}) : \text{Type}$, i.e. $Cs \vdash (\text{forall } y : B \Rightarrow e\{y/x\}) : \text{Type}$. This follows by inversion and the properties of substitution.

Universes

Recall that the type of `abs_prod_type` (with some alpha-renaming) is as follows:

```
abs_prod_type@{UB UE UT UM | UT >= max(UB, UE), UM > UT}:
  forall {B:Type@{UB}} (x : B) (e : Type@{UE}), M@{UM} Type@{UT}
```

Taking into account universes, we have to show that $Cs \vdash (\text{forall } y : B \Rightarrow e\{y/x\}) : \text{Type@}\{UT\}$. It suffices to show that (1) $Cs \vdash$

$B : \text{Type@}\{\text{UT}\}$, and (2) $(\Omega\text{Cs}, \Sigma\text{Cs}, (\Gamma\text{Cs}, y : B)) \vdash e\{y/x\} : \text{Type@}\{\text{UT}\}$. By assumption, we have $\text{Cs} \vdash B : \text{Type@}\{\text{UB}\}$, $\Omega\text{Cs}, \Sigma\text{Cs}, (\Gamma\text{Cs}, y : B) \vdash e\{y/x\} : \text{Type@}\{\text{UE}\}$, and $\text{UT} \geq \max(\text{UB}, \text{UE})$. (1) and (2) follow by cumulativity.

Now we consider some cases when $V \leq \text{Code } e$.

BindR We have $V = \text{Ret } e$, $\text{Cs} = \text{Bind } f :: \text{Cs}'$, and $M' = \text{Code } (f e) :: \text{Cs}'$. By inversion we have $\text{Cs} \vdash \text{Ret } e : B$, $\text{Cs}' \vdash f : B \rightarrow M A$, and $\text{Cs}' : B \Rightarrow A$.

To show: $M' : A$, i.e. (1) $\text{Cs}' \vdash \text{Code } (f e) : B$ and (2) $\text{Cs}' : B \Rightarrow A$.

Obligation (1) follows from $\Omega\text{Cs}' = \Omega\text{Cs}$, $\Sigma\text{Cs}' = \Sigma\text{Cs}$, $\Gamma\text{Cs}' = \Gamma\text{Cs}$, $\text{Cs}' \vdash f : B \rightarrow M A$, and **VCode**. Obligation (2) is true by assumption.

NuR We have $V = \text{Ret } e$ and $\text{Cs} = \text{Nu } (x[:= e'] : B) \cdot \text{Cs}'$.

We need to ensure that the first condition of the following derivation holds (the second one is easy by inversion of the typing of Cs):

$$\frac{\text{Machine} \quad \text{Cs}' \vdash_v \text{Ret } e : C \quad \text{Cs}' : C \Rightarrow A}{\text{Ret } e :: \text{Cs}' : A}$$

We know $\Omega\text{Cs}'; \Sigma\text{Cs}'; \Gamma\text{Cs}', x[:= e'] : B \vdash \text{Ret } e : C$. But since x does not occur in e , and by well-formedness of the context, x also does not occur in $\Gamma\text{Cs}'$, then we can perform strengthening and conclude $\text{Cs}' \vdash e : C$. We have $\Omega_i(M') = \Omega_i(M)$ and $\Sigma_i(M') = \Sigma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

MtryE We have that $V = \text{Raise } e$ and $\text{Cs} = \text{Mtry } (\text{deltau}, \Sigma_\Delta) \cdot f :: \text{Cs}'$. By inversion of typing, $\Sigma_\Delta \oplus \Sigma\text{Cs}'; \Gamma\text{Cs}'; e \vdash \text{Exception} :$ and $\Sigma\text{Cs}'; \Gamma\text{Cs}'; f \vdash \text{Exception} \rightarrow M B :$ and $\text{Cs}' : B \Rightarrow A$. We need to prove that

$$\text{Code } (f e) :: \text{Cs}' : A$$

which essentially boils down to showing that

$$\Omega\text{Cs}'; \Sigma\text{Cs}'; \Gamma\text{Cs}' \vdash \text{Code } (f e) : B$$

which in turn can be proven from

$$\text{Cs}' \vdash e : \text{Exception}$$

This is exactly the hypothesis of the rule.

We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.

MtryR We have $\text{Cs} = \text{Ret } e :: \text{Mtry}(\Omega_\Delta, \Sigma_\Delta) \cdot f :: \text{Cs}'$ and $M' = \text{Ret } e :: (\Omega_\Delta, \Sigma_\Delta) \oplus_F \text{Cs}'$. We also have $\vdash \Omega_\Delta \oplus \Omega\text{Cs}', \Omega_\Delta \oplus \Omega\text{Cs}' \vdash \Sigma_\Delta \oplus \Sigma\text{Cs}'$, and $\Omega_\Delta \oplus \Omega\text{Cs}'; \Sigma_\Delta \oplus \Sigma\text{Cs}'; \Gamma\text{Cs}' \vdash e : B$.

To show: $M' : A$, i.e. (1) $\Omega_\Delta \oplus \Omega\text{Cs}'; \Sigma_\Delta \oplus \Sigma\text{Cs}'; \Gamma\text{Cs}' \vdash e : B$ and (2) $(\Omega_\Delta, \Sigma_\Delta) \oplus_F \text{Cs}' : B \Rightarrow A$. Obligation (1) is one of our assumptions. Obligation (2) follows from **13.5.5**.

The choice of $\Omega_{i,\Delta}$ and $\Sigma_{i,\Delta}$ depends on the shape of Cs' :

- Cs' contains **Mtry** nodes. We have $\Omega_i(M') = \Omega_i(M)$ and $\Sigma_i(M') = \Sigma_i(M)$. We choose $\Omega_{i,\Delta} = (\emptyset, \emptyset)$ and $\Sigma_{i,\Delta} = \emptyset$.
- Cs' does not contain **Mtry** nodes. We have $\Omega_i(M') = \Omega_\Delta \oplus \Omega_i(M)$ and $\Sigma_i(M') = \Sigma_\Delta \oplus \Sigma_i(M)$. We choose $\Omega_{i,\Delta} = \Omega_\Delta$ and $\Sigma_{i,\Delta} = \Sigma_\Delta$.

RemoveV We have $X \in \{\text{Ret}, \text{Raise}\}$, $V = X \ e$, $Cs = \text{Remove } (x : B) \cdot :: Cs'$, and $M' = X \ e :: Cs'$. By inversion, we have $Cs \vdash_v X \ e : B$, $Cs' : B \Rightarrow A$, $\Omega Cs = \Omega Cs'$, $\Sigma Cs = \Sigma Cs'$, $\Gamma Cs = \Gamma_1, \Gamma_2$, $\Gamma Cs' = \Gamma_1, x : B, \Gamma_2$, and $\Omega Cs; \Sigma Cs; \Gamma_1, \Gamma_2 \vdash B : s$.

To show: $M' : A$, i.e. (1) $Cs' \vdash_v X \ e : B$ and (2) $Cs' : B \Rightarrow A$.

Obligation (1) follows from weakening $Cs \vdash_v X \ e : C$. Obligation (2) is true by assumption.

ReplaceV $X \in \{\text{Ret}, \text{Raise}\}$, $V = X \ e$, $Cs = \text{Replace } (x : D) \cdot :: Cs'$, and $M' = X \ e :: Cs'$. By inversion, we have $Cs \vdash_v X \ e : B$, $Cs' : B \Rightarrow A$, $\Omega Cs = \Omega Cs'$, $\Sigma Cs = \Sigma Cs'$, $\Gamma Cs = \Gamma_1, x : D, \Gamma_2$, $\Gamma Cs' = \Gamma_1, x : C, \Gamma_2$, and $Cs \vdash C : s$, as well as $Cs \vdash B : s'$.

To show: $M' : A$, i.e. (1) $Cs' \vdash_v X \ e : B$ and (2) $Cs' : B \Rightarrow A$.

Obligation (1) follows from $Cs \vdash_v X \ e : C$ and compatibility of typing with convertibility in variables' types. Obligation (2) is true by assumption.

□

Lemma 13.5.7 (Auxiliary End-to-End Soundness). *Let $M = V :: C$ such that (a) there exists A , $M : A$, (b) $\vdash \Omega(C); \Sigma(C); \Gamma(C)$, and (c) there exists M' such that $M \xrightarrow{\text{prog}}^* M'$.*

Then $M' : A$ and there exist $\Omega_\Delta, \Sigma_\Delta$ such that $\Omega_i(M') = \Omega_i(M) \oplus \Omega_\Delta$, $\Sigma_i(M') = \Sigma_i(M) \oplus \Sigma_\Delta$, and $\Gamma(M') = \Gamma(M)$.

Proof. By induction on the number of $\xrightarrow{\text{prog}}$ -steps in $M \xrightarrow{\text{prog}}^* M'$.

0 Steps We have $M' = M$. All proof obligations are trivial.

$n + 1$ Steps We have M'' such that $M \xrightarrow{\text{prog}}^* M''$ and $M'' \xrightarrow{\text{prog}} M'$. All obligations follow from II.6 and associativity of $\oplus$.

□

Execution of Mtac2 metaprograms (and tactics) happens in two steps: first, the fully applied metaprogram is typechecked. Then, the program is run in the stack machine. Finally, the result of the machine is used as a proof term for the current goal. (If the metaprogram is a tactic, newly created goals are made visible to the user; see Section 11.). Thus, we formulate the following end-to-end soundness theorem of the Mtac2 interpreter. It assumes that the input is well typed (expressed via the typing judgment on the initial machine) and proves that any final state after execution will be well typed in the initial local context and extensions of the universe and meta-context.

Theorem II.7 (End-to-End Soundness). *Let $M = \text{code } t :: (\Omega, \Sigma, \Gamma, \cdot)$ such that (a) there exists $A, M : A$, (b) $\vdash \Omega; \Sigma; \Gamma$, and (c) there exist Ω', Σ' such that $M \xrightarrow{\text{prog}}^* V :: (\Omega', \Sigma', \Gamma', \cdot)$ and $V \neq \text{code } _$.*

Then there exist $\Omega_\Delta, \Sigma_\Delta$ such that $\Omega' = \Omega \oplus \Omega_\Delta$, $\Sigma' = \Sigma \oplus \Sigma_\Delta$, and $\Gamma' = \Gamma$, and either

1. *there exists e such that $V = \text{Ret } e$ and $\Omega'; \Sigma'; \Gamma \vdash e : A$, or*
2. *there exists e such that $V = \text{Fail } e$ and $\Omega'; \Sigma'; \Gamma \vdash e : \text{Exception}$.*

Proof. Follows from 13.5.7. □

14

Technical details concerning some new features of Mtac2

This section presents a subset of the most important changes introduced in Mtac2.

14.1 POLYMORPHIC UNIVERSES

To allow meta-meta-programming, *i.e.*, Mtac2 programs that construct Mtac2 programs, the monad $\mathbb{M}$ in Mtac2 was declared *universe polymorphic* [ST14].

While Rocq supports universe polymorphism, its standard library still uses an older mechanism called template universe polymorphism for all inductive types defined therein. Template universe polymorphism poses a significant challenge because it effectively punishes generic programming: uses of generic datatypes (such as `list`) that do not instantiate all their arguments result in global universe constraints on all further instances of `list`. (See Section 17 for a detailed explanation.) As a consequence, this mechanism interacts very badly with the kind of generic programming that happens not just in Mtac2 meta-programs but also, for example, in Rocq libraries defining Haskell-style type classes such as `FMap`, and instances thereof for standard datatypes.

As a result, Mtac2 could not be implemented on top of Rocq’s standard library. While the Rocq developers are discussing moving the standard library away from template universe polymorphism¹, Mtac2 now ships with a properly universe-polymorphic copy of a very small subset of the standard library used exclusively for implementing tactics and meta-programs. In fact, when we mentioned product types, *e.g.*, `A * list goal` in Section 11, we introduced a white lie: in the code we actually employ a distinct, bespoke notation for Mtac2’s product type.

See Section 17 for a detailed discussion of universes in Mtac2 and the challenges they pose.

[ST14] Sozeau et al., “Universe Polymorphism in Coq”

¹ <https://github.com/coq/coq/issues/6093>, Rocq Pull Request #6093: Future of template polymorphism.

14.2 SORT POLYMORPHISM

The constructor `Metavar` as presented in Section 10.3 has a limitation. Note the type `A` of the meta-variable: it has (implicitly) the `Type` sort. Since Rocq includes a infinite hierarchy of types, it is always possible to *up-cast* any sort to a `Type` having a large enough *universe level* (implicit in the `Type` keyword). However, in doing so we miss some information: if `A` were instantiated with a `Prop`, a sort in which *impredicativity* holds, then we cannot use this property, since `Type` is *predicative*. Figure 14.1 shows an example of this issue: in the definition `idP`, the identity function’s type is in `Prop`, while in `idT` we up-cast it to `Type` (note the notation `P : Type` for the returning type). With the former, we can instantiate it with itself, while in the latter we cannot.

Figure 14.1: An example of why up-casting `Prop` to a `Type` breaks impredicativity.

```
Definition idP {P:Prop} (x:P) : P := x.
Definition impredP := idP (@idP).

Definition idT {P:Prop} (x:P) : P : Type := x.
Fail Definition impredT := idT (@idT).
```

Mtac2’s goals as presented so far are lacking a way to distinguished a goal in the impredicative `Prop` sort from one with the predicative `Type` sort. One simple option is to add another constructor, say, `MetavarP`. However, this leads to a horrible duplication of code, which has to deal with each case accordingly, even if the sort plays no role in the tactic. Instead, we decided to employ the powerful type system we inherit from Rocq, and introduce a `Sort` type and make `Metavar`’s type be dependent on it:

```
Metavar : forall (s : Sort) {A : stype_of s}, selelem_of A -> goal
```

Figure 14.2: The `Sort` type and some helper functions.

```
Inductive Sort : Type := S_Type | S_Prop.

Definition stype_of (s : Sort) : Type :=
  match s with S_Type => Type | S_Prop => Prop end.

Definition selelem_of {s : Sort} : stype_of s -> Type :=
  match s return stype_of s -> Type with
  | S_Type => fun x:Type => x
  | S_Prop => fun x:Prop => x
  end.

Coercion stype_of : Sort ->> Sortclass.
Coercion selelem_of : stype_of ->> Sortclass.
```

Figure 14.2 shows the `Sort` type together with the two functions needed by the `Metavar` constructor. In a nutshell, `Sort` just enumerates the two sorts handled by Mtac2,² with self-descriptive names (`S_Prop` and `S_Type`); `stype_of s` returns the corresponding sort, according to `s`; and `selelem_of A` allows treating `A : stype_of s` as a type: it represents the elements inhabiting `A`.

As common when dealing with dependently-typed pattern matching, in the definition of `selelem_of` we are using the *convoy pattern* [Chl11,

² Rocq also has support for a predicative `Set` sort. However, it can be encoded with a `Type` without loss of generality.

[Chl11] Chlipala, “Certified Programming with Dependent Types”

Chp. 8], in which we embed the dependently-typed types as part of the return type of the pattern matching construct. It is interesting to note that in this function we are up-casting whatever sort `s` is to `Type`. However, that is not a problem in this case, since when we work with elements of `A : stype_of s` we will perform pattern-matching like in Line 7 of Figure 14.2, therefore effectively *down-casting* it to the corresponding sort.

As an example, the function `sorted_evar` in Figure 14.3 shows how to apply the `Sort` type to create a meta-variable with the right sort. In this code we are using the two coercions listed at the end of Figure 14.2: when we write `(T : s)`, Rocq infers `(T : stype_of s)`, and when we write `T` in the place where a `Type` is expected—as in `eval T`—Rocq infers `selem_of T`.

```

1 Definition sorted_evar (s: Sort) : forall (T : s), M T :=
2   match s with
3   | S_Prop => fun (T : Prop) => eval T
4   | S_Type => fun (T : Type) => eval T
5   end.

```

Figure 14.3: Example of use of the `Sort` type.

SProp

The `SProp` sort [Gil+19] that has recently been added to Rocq is not yet supported by `Mtac2`’s sort polymorphism.

[Gil+19] Gilbert et al., “Definitional proof-irrelevance without K”

14.3 OPAQUE DEFINITIONS

In `Mtac1` the primitives of the language were defined as constructors of the inductive type `M`. However, this design had two issues:

1. Universes: When `M` was made universe polymorphic (see Section 14.1), it became parameterized over the union of all universes required by its constructors. As a consequence, even simple primitives such as `ret`—which itself only requires at most two universes—were forced to carry a prohibitively large number of universes.
2. Fixed-points: As described in the original paper [Zil+13], the fixed-point operators of `Mtac1` had a dummy subset predicate in order to circumvent the *positivity restriction* of Rocq. This predicate would then be instantiated with the identity function. Though not a problem in practice, it added a seemingly unnecessary indirection.

[Zil+13] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

In `Mtac2`, we solve both issues by taking inspiration from the `Exception` type (see Section 9) and making the `M` type into a trivial inductive type:

```
Inductive M : Type -> Prop := mkt : forall {A}, M A.
```

Primitives can then be defined as opaque wrapper definitions around the constructor `mkt`. As an example, we give below the definition of `Mtac2`’s unary fixed-point combinator. Note the negative occurrence of `M`, which no longer requires any indirection.

```
1 Definition fix1 : forall {A: Type} (B: A->Type),  
2   ((forall x: A, M (B x)) -> (forall x: A, M (B x))) ->  
3   forall x: A, M (B x).  
4   intros; constructor. Qed.
```

15

Related and future work

The original Mtac paper [Zil+15] includes a detailed comparison between the core of Mtac and other languages for metaprogramming in proof assistants, such as Beluga [Pie08] and VeriML [SS10]. We focus here on tactic languages for backward reasoning and on recent metaprogramming techniques related to type theory.

Tactic languages with a deep embedding of the logic.

In recent years there is a trend among interactive theorem provers: provide a high-level metalanguage containing a deep embedding of the logic, with a mechanism to quote/unquote terms to/from the deeply-embedded representation [Ebn+17; Ana+18; CB16; Agd18; Mar+19]. This enables programming of tactic primitives without having to resort to the language in which the prover is built (e.g., OCaml in the case of Rocq).

A major limitation of deep-embedding approaches is that meta-programs and tactics cannot be given meaningful types to describe their behavior in the way that Mtac2 tactics can (thanks to Mtac’s shallow embedding of terms). Also, the deep-embedding approaches must pay the cost of quoting/unquoting of terms, and must include ad-hoc features to parse and print nicely quoted/unquoted terms. To give a sense of the problem, in Idris a meta-function that determines if a given term corresponds to the number zero can be written as:

```
isZeroR : TT -> Maybe Bool
isZeroR (P _ (NS (UN "Z") ["Nat", "Prelude"])) _ =
  Just True
isZeroR (App (P _ (NS (UN "S") ["Nat", "Prelude"])) _ n) =
  Just False
isZeroR _ =
  Nothing
```

Note the constructors of the TT datatype representing terms in the language: App (for application), P (for constructor). Thanks to the quoting mechanism devised in [Chr14], the complexity of dealing with bare terms can be hidden using the surface language of the prover:

```
isZeroR : TT -> Maybe Bool
isZeroR `(Z) = Just True
isZeroR `(S ~n) = Just False
isZeroR _ = Nothing
```

[Zil+15] Ziliani et al., “Mtac: A monad for typed tactic programming in Coq”

[Pie08] Pientka, “A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions”

[SS10] Stampoulis et al., “VeriML: Typed computation of logical terms inside a language with effects”

[Ebn+17] Ebner et al., “A metaprogramming framework for formal verification”

[Ana+18] Anand et al., “Towards Certified Meta-Programming with Typed Template-Coq”

[CB16] Christiansen et al., “Elaborator reflection: Extending Idris in Idris”

[Agd18] Agda Development Team, “Reflection”

[Mar+19] Martínez et al., “Meta-F*: Proof automation with SMT, tactics, and metaprograms”

[Chr14] Christiansen, “Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection”

In contrast, the same function can be coded in Mtac as:

```
Definition isZeroNR {T} (t: T) : M (option bool) :=
  mmatch T with
  | nat : Type => [eqT]
    let t1 := internal_meq_rew Type T _ t nat eqT in
    match t1 with
    | 0 => ret (Some true)
    | S _ => ret (Some false)
    end
  | _ => ret None
end.
```

This code has two advantages: the term t is not quoted, and once we know we have an element of `nat`, we can perform an *exhaustive* pattern-matching on t . The `let`-binding allows us to cast t to an element of type `nat`, using the equality witnessed by `eqT`. If we do not care about the pattern being exhaustive, we can resort to a more direct encoding of the function by using Σ -types:

```
Definition isZeroNR' {T} (t: T) : M (option bool) :=
  mmatch Dyn t with
  | Dyn 0 => ret (Some true)
  | [? n] Dyn (S n) => ret (Some false)
  | _ => ret None
end.
```

One advantage of deep-embedding approaches is that they are more flexible, as they permit the construction of arbitrary terms (even ill-typed ones).

Among deep-embedding approaches there are two broad classes, according to whether the metalanguage for tactic programming is the base logic of the prover or a different language. The former class includes Idris [CB16], Agda [Agd18], MetaRocq [Ana+18] formerly known as Template Rocq, and Meta-F* [Mar+19]. The advantages of reusing the base logic of the prover—advantages that Mtac shares—are that 1) the user is not forced to learn a new language, and 2) one has access to the full set of tools already constructed in the prover to build programs, including type classes and any existing meta-programming tools (including Mtac2 itself, which can itself be used to construct Mtac2 programs).

While Idris, Agda, and Rocq are all languages close to type theory, F*, the base language used by Meta-F* is quite different. It is a language for program verification using extensible effects, and therefore Meta-F*'s tactics are defined as just another effect. Similar to the other languages, it includes quotation/antiquotation primitives in order to inspect terms. The effect of a tactic is, naturally, on the proof state, which—similar to Mtac2's—is backtracked at each failure. Unlike Mtac2, and like Ltac, it is possible to code tactics that fail late (see Section 10.6).

The class of deeply-embedded tactic languages using a different metalanguage includes Ltac2 [Péd19] and Lean [Ebn+17], which provide bespoke metalanguages for writing tactics. Ltac2 is a new tactic language intended as a replacement for Ltac. It has ML-style syntax, typing, and semantics, and provides a large (and extensible) API, with access to the internals of Rocq. Lean's meta-language resembles that of the logic,

[CB16] Christiansen et al., “Elaborator reflection: Extending Idris in Idris”

[Agd18] Agda Development Team, “Reflection”

[Ana+18] Anand et al., “Towards Certified Meta-Programming with Typed Template-Coq”

[Mar+19] Martínez et al., “Meta-F*: Proof automation with SMT, tactics, and metaprograms”

[Péd19] Pédrot, “Ltac2: Tactical Warfare”

[Ebn+17] Ebner et al., “A metaprogramming framework for formal verification”

with a special keyword `meta` to distinguish meta-programs from regular programs. It includes several interesting features, such as SMT integration and a debugging API, and it performs incredibly well according to the benchmarks reported by [Ebn+17].

Tactic languages as DSLs.

[Bow16] discusses *Cur*, a small dependently-typed language incorporating language extension early on in the development process. Then, every feature of the language—including a tactic language—can be constructed as an extension. In [Cha+18] the authors conjecture it is possible to code *Mtac* in their framework. An alternative that we consider interesting to explore in the future is to build the extension mechanism as part of the *M* monad, and then build the outermost tools of the proof assistant on top of it: the notation mechanism, type inference, etc.

[Bow16] Bowman, “Growing a Proof Assistant”

[Cha+18] Chang et al., “Dependent Type Systems as Macros”

Tactic languages with their semantics encoded in the prover’s logic.

Rtac [MB16] provides reflection-based tactics in pure Gallina (the base language of *Rocq*). As with *Mtac2*, an *Rtac* tactic is simply a *Rocq* term. Yet, contrary to *Mtac2*, the semantics of *Rtac* stays inside *Rocq*. (There is no external interpreter.) Therefore, while *Mtac2*’s interpreter must produce a concrete proof term to serve as a trusted proof, a tactic application in *Rtac* is in itself a proof, and therefore it must not contain any effectful computation in it. The execution of reflective tactics is usually faster than checking a large proof term, although the proof search process of the tactic has to be executed twice, once for constructing the proof, and once after the proof is complete and checked by *Rocq*’s kernel.

[MB16] Malecha et al., “Extensible and efficient automation through reflective tactics”

Both limitations, the lack of effects and the need for replaying the proof, are not necessarily restrictions, as the *Cybele* framework shows [Cla+13]. Like *Rtac*, the *Cybele* framework allows the construction of reflective tactics for *Rocq*, but it permits certain effects—like non-termination. Under the hood, a reflective tactic expects information to mimic the effect—like the number of steps. This information, called a *prophecy*, is obtained from an external oracle, which is nothing more than a variation of the tactic itself, which is extracted to OCaml, compiled, and executed. The prophecy provides enough information to safely replay the reflective tactic inside *Rocq*. The cost of searching a successful path in the proof search process is paid only once, in the execution of the compiled tactic. When replaying the proof inside *Rocq*, the prophecy includes enough information to just execute the known successful branches.

[Cla+13] Claret et al., “Lightweight proof by reflection using a posteriori simulation of effectful computation”

That said, a fundamental limitation that *Rtac* and *Cybele* share is that their tactics are required to be statically proven sound, meaning that their computation must *always* yield a valid proof. This is in contrast to *Mtac2*, which allows the use of dynamically-checked meta-programming constructs such as its primitives `nu` and `abs_fun`. As a consequence, the construction of tactics in *Rtac* and *Cybele* requires a significant amount of boilerplate ensuring their unconditional correctness. To alleviate this burden, *Rtac* provides soundness proofs for basic tactics, and an *Ltac*

tactic to automatically solve the user’s tactic’s soundness proof built on top of them. Another limitation of *Rtac* is that it lacks support for goals with dependent types. As our case study routinely deals with dependently-typed goals, we believe this to be a major restriction.

Shallowly-embedded tactic languages.

[Cau18] Cauderlier, “Tactics and certificates in Meta Dedukti”

[Boe+12] Boespflug et al., “Dedukti: A universal proof checker”

[Cau18] built a tactic language similar to *Mtac* for *Dedukti* [Boe+12]. An interesting aspect of this work lies in its use of *Dedukti*’s rewrite rules to build the interpreter, thereby enabling the construction of language primitives directly within the prover. Then, to execute tactics, it uses *Meta Dedukti*, a tool for *Dedukti* that normalizes terms according to a rewrite theory (in this case, the interpreter of tactics). Unlike with *Rtac* and *Cybele*, there is no need to justify the soundness of the rules inside the language because, as in *Mtac*, *Dedukti* typechecks the output of a tactic after the fact.

Like with *Mtac2*, [Cau18] also includes a type for tactics. That type, however, does not return a list of goals but a single goal instead. As a consequence, tactics that operate on multiple subgoals need to take a corresponding number of continuations as arguments, instead of simply composing the tactic with a list of tactics as is done in *Mtac2*. Consider, for example, a proof script that splits a conjunction $P \wedge Q$, applies two tactics *tacP* and *tacQ* to their respective subgoals, and then applies *tacB* to both subgoals. In *Mtac2*, we can write this as `split &> [m: tacP | tacQ] &> tacB`. In the system presented by [Cau18], however, this is written roughly as `split (tacP; tacB) (tacQ; tacB)` (where we use `;` to signify the `bind` operation). Note that the *tacB* tactic has to be repeated in both continuations since there is no way to operate on two subgoals with the same tactic.

When compared to *Mtac*, the expressiveness of [Cau18] is somewhat limited. For a start, *Meta Dedukti* has no notion of meta-variables, and they cannot support the kind of dynamic checks we use in *Mtac*, which limits the kind of primitives that can be encoded. For instance, in *Mtac* we provide primitives for introducing terms in the context (the `nu` primitive) and for creating an abstraction (`abs_fun`). The former dynamically checks that the variable does not escape the context. This check cannot be coded directly in *Meta Dedukti* and forces a restricted form of operator which combines the effect of `nu` and `abs_fun`, called `mintro`.

Tactic languages with dynamic enforcement of invariants.

[GKL13] Grov et al., “A Graphical Language for Proof Strategies”

[Lin+16] Lin et al., “A Super Industrial Application of *PSGraph*”

In *PSGraph* [GKL13] the authors make a strong case against fragile tactics, in particular in the context of industrial applications [Lin+16] where tactics take several pages of code with a hard-to-understand control flow. Instead of proposing a typed approach like *Mtac2*’s, they use two interesting ingredients: a graphical tool to help construct the control flow of tactics, together with dynamic checks at each expected input of a tactic, to ensure tactics are applied to the right goal. We strongly believe such tool can be successfully applied in the context of *Mtac2*: The dynamic

nature of the tactic type introduced in Section 10 allows to introduce such dynamic checks. Furthermore, by *promoting* goals as in Section 11 we can mix such checks with static typing, something not considered in the aforementioned works.

A function type for backwards tactics.

The type for tactics presented in Section 14 closely resembles the type that LCF [GMW79] introduced to handle backward reasoning, and is more or less the type that modern proof assistants adopted. However, the original type in LCF shares no information about partial proofs (*i.e.*, meta-variables). As noted by [Asp+09], this omission limits the expressiveness of tactics, and for this reason, most modern proof assistants enrich the type to include a context for meta-variables, as well as other relevant information about the proof state.

In Rocq, for instance, it used to be the case that tactics explicitly received and returned the context of meta-variables (`metavars`), having the type: `goal * metavars -> goal list * metavars`. However, this type was not rich enough to support tactics operating on multiple goals. Therefore, it has since been replaced with a monadic type that, similar to `Mtac2`, embeds the context of meta-variables. Additionally, and unlike `Mtac2`, it also embeds information about the *focused* goals (a set of goals currently under consideration). This approach was developed by [Spi10] and is beautifully summarized by [Péd16a]. It would be interesting to study if this enriched type could be encoded in `Mtac2` as is, or if it will require further support from the interpreter.

[GMW79] Gordon et al., “Edinburgh LCF”

[Asp+09] Asperti et al., “A new type for tactics”

[Spi10] Spiwack, “An abstract type for constructing tactics in Coq”

[Péd16a] Pédrot, “Ticking like a clockwork: The new Coq tactics”

15.1 FUTURE WORK

There are several lines for improving `Mtac2`. Here we include what we think are most relevant ones.

Soundness proof in Rocq

Having written (and rewritten) several times the proof of soundness of `Mtac` (1 and 2), we would like to have the proof written in Rocq. The most reasonable path to do it would be to code `Mtac`’s interpreter in `MetaRocq` [Ana+18], which already contains a lot of the infrastructure to reason about Rocq’s plugins in Rocq.

[Ana+18] Anand et al., “Towards Certified Meta-Programming with Typed Template-Coq”

Extending the language

There are several interesting additions to the language, some of which are already being tested: dynamic creation of definitions, interfacing with `Ltac/Ltac2` tactics, dynamic creation of notations, typed decomposition of terms (see below), etc.

Performance

The performance characteristics of Mtac2 are influenced by different, sometimes non-obvious factors. In microbenchmark, Mtac2 generally outperforms all other safe metaprogramming languages and is only surpassed by unsafe Ltac2, TemplateRocq, and bespoke OCaml plugins. Thorough results from a reification benchmark can be found in [Gro21]. However, in larger examples, Mtac2 suffers from performance issues that are outside of its control. The most important factor is Mtac2's use of universe polymorphism. Part III provides a detailed discussion of the key factors that affect Mtac2's performance.

[Gro21] Gross, "Performance Engineering of Proof-Based Software Systems at Scale"

Bibliography

- [Agd18] The Agda Development Team. “Reflection”. 2018. URL: <https://agda.readthedocs.io/en/v2.5.3/language/reflection.html> (cit. on pp. 181, 182).
- [Ana+18] Abhishek Anand, Simon Boulrier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. “Towards Certified Meta-Programming with Typed Template-Coq”. In: *Interactive Theorem Proving*. Ed. by Jeremy Avigad and Assia Mahboubi. Cham: Springer International Publishing, 2018, pp. 20–39. ISBN: 978-3-319-94821-8 (cit. on pp. 181, 182, 185).
- [Asp+09] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. “A new type for tactics”. In: *PLMMS* (2009), p. 22 (cit. on p. 185).
- [Bar09] Bruno Barras. “The Dark Side of Ltac”. 2009. URL: <https://coq.inria.fr/files/adt-30jun09-bruno-ltac.pdf> (cit. on p. 109).
- [Boe+12] Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. “Dedukti: A universal proof checker”. In: *Journées communes LTP - LAC*. 2012. URL: <https://hal-mines-paristech.archives-ouvertes.fr/hal-01537578> (cit. on p. 184).
- [Bow16] William J. Bowman. “Growing a Proof Assistant”. In: *Workshop on Higher-order Programming with Effects (HOPE 2016)*. 2016 (cit. on p. 183).
- [Cau18] Raphaël Cauderlier. “Tactics and certificates in Meta Dedukti”. In: *ITP*. ITP ‘18. 2018 (cit. on p. 184).
- [Cha+18] Stephen Chang, Michel Ballantyne, Marcela Poffald, and William J. Bowman. “Dependent Type Systems as Macros”. 2018 (cit. on p. 183).
- [Chl11] Adam Chlipala. “Certified Programming with Dependent Types”. <http://adam.chlipala.net/cpdt/>. MIT Press, 2011. URL: <http://adam.chlipala.net/cpdt/%7D> (cit. on pp. 119, 178).
- [CB16] David Christiansen and Edwin Brady. “Elaborator reflection: Extending Idris in Idris”. In: *ICFP*. 2016, pp. 284–297. DOI: [10.1145/2951913.2951932](https://doi.org/10.1145/2951913.2951932) (cit. on pp. 181, 182).
- [Chr14] David Raymond Christiansen. “Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection”. In: *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*. IFL ’14. Boston, MA, USA: ACM, 2014, 1:1–1:9. ISBN: 978-1-4503-3284-2. DOI: [10.1145/2746325](https://doi.org/10.1145/2746325).

2746326. URL: <http://doi.acm.org/10.1145/2746325.2746326> (cit. on p. 181).
- [Cla+13] Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. “Lightweight proof by reflection using a posteriori simulation of effectful computation”. In: *ITP*. Vol. 7998. LNCS. 2013, pp. 67–83. DOI: [10.1007/978-3-642-39634-2_8](https://doi.org/10.1007/978-3-642-39634-2_8) (cit. on p. 183).
- [Ebn+17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. “A metaprogramming framework for formal verification”. In: *PACMPL* 1.ICFP (2017), 34:1–34:29. DOI: [10.1145/3110278](https://doi.org/10.1145/3110278) (cit. on pp. 181–183).
- [FH92] Matthias Felleisen and Robert Hieb. “The revised report on the syntactic theories of sequential control and state”. In: *TCS* 103.2 (1992), pp. 235–271. DOI: [10.1016/0304-3975\(92\)90014-7](https://doi.org/10.1016/0304-3975(92)90014-7) (cit. on p. 133).
- [Gil+19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. “Definitional proof-irrelevance without K”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 3:1–3:28 (cit. on p. 179).
- [GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. “Edinburgh LCF”. Vol. 78. LNCS. 1979. DOI: [10.1007/3-540-09724-4](https://doi.org/10.1007/3-540-09724-4) (cit. on p. 185).
- [GL02] Benjamin Grégoire and Xavier Leroy. “A compiled implementation of strong reduction”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP ’02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. 2002, pp. 235–246. DOI: [10.1145/581478.581501](https://doi.org/10.1145/581478.581501). URL: <https://doi.org/10.1145/581478.581501> (cit. on p. 155).
- [Gro21] Jason S. Gross. “Performance Engineering of Proof-Based Software Systems at Scale”. PhD Thesis. Massachusetts Institute of Technology, Feb. 2021. URL: <https://jasongross.github.io/papers/2021-JGross-PhD-EECS-Feb2021.pdf> (cit. on p. 186).
- [GKL13] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. “A Graphical Language for Proof Strategies”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 324–339. ISBN: 978-3-642-45221-5 (cit. on p. 184).
- [Jun+18a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “RustBelt: Securing the foundations of the Rust programming language”. In: *PACMPL* 2.POPL (2018), 66:1–66:34. DOI: [10.1145/3158154](https://doi.org/10.1145/3158154) (cit. on p. 98).

- [Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *JFP* 28.e20 (Nov. 2018) (cit. on p. 132).
- [Kai+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. “Strong logic for weak memory: Reasoning about release-acquire consistency in Iris”. In: *ECOOP*. Vol. 74. LIPIcs. 2017, 17:1–17:29. DOI: [10.4230/LIPIcs.ECOOP.2017.17](https://doi.org/10.4230/LIPIcs.ECOOP.2017.17) (cit. on p. 98).
- [Kai+18] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. “Coq repository for Mtac2”. <https://github.com/Mtac2/Mtac2/>. 2018 (cit. on p. 102).
- [KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive Proofs in Higher-Order Concurrent Separation Logic”. In: *POPL*. 2017, pp. 205–217. DOI: [10.1145/3093333.3009855](https://doi.org/10.1145/3093333.3009855) (cit. on p. 98).
- [Lin+16] Yuhui Lin, Gudmund Grov, Colin O’Halloran, and Priya G. “A Super Industrial Application of PSGraph”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro. Cham: Springer International Publishing, 2016, pp. 319–325. ISBN: 978-3-319-33600-8 (cit. on p. 184).
- [MB16] Gregory Malecha and Jesper Bengtson. “Extensible and efficient automation through reflective tactics”. In: *ESOP*. Vol. 9632. LNCS. 2016, pp. 532–559. DOI: [10.1007/978-3-662-49498-1_21](https://doi.org/10.1007/978-3-662-49498-1_21) (cit. on p. 183).
- [Mar+19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. “Meta-F*: Proof automation with SMT, tactics, and metaprograms”. In: *ESOP*. 2019 (cit. on pp. 181, 182).
- [MP08] Conor McBride and Ross Paterson. “Applicative programming with effects”. In: *JFP* 18.1 (2008), pp. 1–13. DOI: [10.1017/S0956796807006326](https://doi.org/10.1017/S0956796807006326) (cit. on p. 127).
- [NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual modal type theory”. In: *ACM Transactions on Computational Logic (TOCL)* 9.3 (2008), 23:1–23:49. DOI: [10.1145/1352582.1352591](https://doi.org/10.1145/1352582.1352591) (cit. on p. 110).
- [Péd16a] Pierre-Marie Pédrôt. “Ticking like a clockwork: The new Coq tactics”. 2016. URL: <http://web.archive.org/web/20210418210315/http://coqhott.gforge.inria.fr/blog/coq-tactic-engine> (cit. on p. 185).

- [Péd16b] Pierre-Marie Pédro. “Towards Ltac 2.0”. DeepSpec Workshop. 2016 (cit. on p. 109).
- [Péd19] Pierre-Marie Pédro. “Ltac2: Tactical Warfare”. The Fifth International Workshop on Coq for Programming Languages. 2019 (cit. on pp. 109, 182).
- [Pie08] Brigitte Pientka. “A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions”. In: *POPL*. 2008, pp. 371–382. DOI: 10.1145/1328438.1328483 (cit. on p. 181).
- [SO08] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *TPHOLs*. Vol. 5170. LNCS. 2008, pp. 278–293. DOI: 10.1007/978-3-540-71067-7_23 (cit. on p. 118).
- [ST14] Matthieu Sozeau and Nicolas Tabareau. “Universe Polymorphism in Coq”. In: *ITP*. Vol. 8558. LNCS. 2014, pp. 499–514. DOI: 10.1007/978-3-319-08970-6_32 (cit. on p. 177).
- [Spi10] Arnaud Spiwack. “An abstract type for constructing tactics in Coq”. In: *Proof Search in Type Theory*. 2010 (cit. on pp. 119, 185).
- [SS10] Antonis Stampoulis and Zhong Shao. “VeriML: Typed computation of logical terms inside a language with effects”. In: *ICFP*. 2010, pp. 333–344. DOI: 10.1145/1863543.1863591 (cit. on p. 181).
- [SGD17] David Swasey, Deepak Garg, and Derek Dreyer. “Robust and compositional verification of object capability patterns”. In: *PACMPL 1.OOPSLA (2017)*, 89:1–89:26. DOI: 10.1145/3133913 (cit. on p. 98).
- [The24] The Coq Development Team. “The Coq Reference Manual – Release 8.19.0”. <https://coq.inria.fr/doc/V8.19.0/refman>. 2024 (cit. on p. 165).
- [Tim+18] Amin Timany, Léo Stefanescu, Morten Krogh-Jespersen, and Lars Birkedal. “A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST”. In: *PACMPL 2.POPL (2018)*, 64:1–64:28. DOI: 10.1145/3158152 (cit. on p. 98).
- [Zil15] Beta Ziliani. “Interactive Typed Tactic Programming for the Coq Proof Assistant”. PhD thesis. MPI-SWS and University of Saarland (UdS), 2015 (cit. on p. 152).
- [Zil+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. “Mtac: A monad for typed tactic programming in Coq”. In: *ICFP*. 2013, pp. 87–100. DOI: 10.1145/2500365.2500579 (cit. on pp. 98, 179).

- [Zil+15] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. “Mtac: A monad for typed tactic programming in Coq”. In: *JFP* 25.e12 (2015). DOI: [10.1017/S0956796815000118](https://doi.org/10.1017/S0956796815000118) (cit. on pp. 95, 98, 103, 106, 146, 150, 181).
- [ZS17] Beta Ziliani and Matthieu Sozeau. “A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading”. In: *JFP* 27 (2017). DOI: [10.1017/S0956796817000028](https://doi.org/10.1017/S0956796817000028) (cit. on pp. 120, 152, 159).

Part III

Mtac2: Engineering Challenges

The work on Mtac2's typed tactics demonstrates that the implementation of tactics can benefit tremendously from Rocq's existing strong type system. However, Mtac2 suffers from several issues in Rocq's implementation. These issues can lead to prohibitive performance problems both in the application of tactics as well as in the design of new tactics. In this part of the thesis, I go into detail about the most important issues that have been holding back Mtac2 development and usage.

16

The Interpreter

The interpreter of Mtac2 is at the heart of the OCaml plugin that implements the Mtac2 language. As Mtac2 consists of primitive operations (primitives) that are used to implement more complex features, the performance of the interpreter is a central piece of Mtac2’s overall performance profile. The interpreter’s implementation largely follows the formalization in Section 13. In particular, the interpreter is implemented by a stack machine.

The kinds of stack items correspond to those shown in Section 13 with one notable difference: `MTry` nodes used for exception handling do not remember differential contexts— instead, they remember the full contexts. This design choice significantly simplifies the implementation. Due to OCaml’s use of garbage collection and the fact that these contexts are functional data structures there is very little memory consumption overhead to keeping the full contexts around.

As far as the performance of Mtac2 is concerned, the most important aspect is the fact that Mtac2 relies on Rocq’s kernel reduction machine. Unlike other users of the machine, Mtac2 has no easy way to improve on the performance of the reduction machine by using Rocq’s virtual machine or native compilation mechanisms. The reason for this is that Mtac2 lacks a syntactic distinction between meta- and object-level terms. Either of these two compilation mechanisms would end up performing full reduction in object terms. This must not happen in a metaprogramming language.

Aside: Call-By-Name

A consequence of this lack of distinction is that Mtac2’s semantics are necessarily call-by-name. This is easy to see in a typical metaprogramming example that relies on the object-level value being in the exact state provided by the user.

```
Definition is_plus (n : nat) : M bool :=  
  mmatch n with  
  | [? x y] (x + y) => M.ret true  
  | _ => M.ret false  
end.
```

```
Mtac2 Run is_plus (1+2). (* Should return [true] *)
```

The fact that `is_plus (1+2)` should return `true` is uncontroversial and directly implies that `is_plus` must be evaluated before its argument.

It is not clear that this limitation is inherent to Mtac2’s approach of re-using the object level type system for its meta language. One could imagine a type theory that provides a modality which blocks terms from being evaluated.¹ However, changing the type theory of Rocq is outside the scope of this thesis and also, arguably, antithetical to the ethos of the Mtac project itself.

¹ The author is not aware of an existing or proposed solution that plays nicely with the dependent nature of Mtac2 metaprograms.

16.1 REDUCTION MACHINE

As described in the semantics, Code stack items are reduced via weak-head reduction (without reducing let-bindings) to expose the next Mtac2 primitive. Due to this reliance on reduction, Mtac2 performance is directly tied to that of Rocq’s weak-head reduction. In particular, Mtac2 relies on Rocq’s `CClosure` module—often referred to as *kernel reduction* because of its use in Rocq’s kernel.

Over the course of investigating the performance problems observed in realistic tactics, there were two measurable performance concerns in Mtac2’s use of the `CClosure` module. At the time of writing, the more impactful issue has already been fixed in Rocq itself.

State-Dependent Caches

The first problem is that Mtac2 does not and cannot reuse `CClosure`’s caching mechanisms across calls to weak-head reduction. `CClosure` automatically caches the result of looking up global and local definitions as well the solutions of existential variables. However, the cache is only valid in the current environment and `evvar` state. In general, Mtac2 code can change the environment and manipulate existential variables and thus the interpreter cannot reuse the caches across most calls to `CClosure`’s reduction functions.

It is still the case, however, that unfolding global definitions could benefit from a persistent cache. Performance profiles show that cache misses make up a noticeable fraction in micro-benchmarks which use neither local definitions nor existential variables. This suggests that performance could be improved by finer-grained control over these caches.

Universe-Dependent Caches

The second problem is also related to caching: `CClosure` cached results of lookups not just by name but by the name paired with its *universe instance*, i.e., the universes used to instantiate the definition’s universe binders. Mtac2’s use of universe polymorphism often defeated the usefulness of `CClosure`’s caches even in individual calls because most uses of the same universe-polymorphic definition will have different universe instances.

As a result of these two issues, time spent in reduction was a bigger factor in the performance profile of Mtac2 than expected. This second issue has been resolved recently[GKY22] and improved Mtac2 runtime significantly. It is still the case, however, that a lot of Mtac2’s runtime is spent in reduction. At the time of writing, it is unclear how much of that is overhead due to implementation choices in CClosure and how much of it could be avoided by using a bespoke reduction mechanism. Furthermore, there is still a significant impact from universes (unrelated to the universe-related caching issue), as we detail in the next chapter.

[GKY22] Gilbert et al., “Delay Universe Substitution in CClosure #15807”

17

Universes

17.1 BACKGROUND: UNIVERSES IN ROCQ

Rocq's type theory features a cumulative hierarchy of universes to avoid the inconsistency of $\text{Type} : \text{Type}$ (as witnessed by Girard's paradox). As a consequence, every mention of Type in Rocq needs to live at a particular *universe level*. The lowest type level is Type_0 , which is called Set in Rocq. To alleviate the burden of managing universes, Rocq takes care to insert fresh universe levels automatically and also manages the necessary constraints to make sure that the current module is consistent.

For example, consider the inductive type wrap defined below. We ignore the Unset command for now.

```
Unset Auto Template Polymorphism.  
Monomorphic Inductive wrap (T:Type) : Type :=  
| Wrap : T -> wrap T.
```

Rocq translates this declaration as follows.

```
Inductive wrap (T : Type@{wrap.u}) : Type@{wrap.u} :=  
| Wrap : T -> wrap T.
```

The universe level wrap.u has been automatically introduced by Rocq. It is a *global universe level*, collected in the *global universe graph* that encompasses the universe levels of all objects loaded in Rocq.

Global universes can have constraints between them. Rocq ensures that the set of all constraints on global universes is always satisfiable. wrap itself introduces a single constraint which is common to all global universes: $\text{Set} < \text{wrap.u}$, i.e., wrap.u is always strictly larger than Set .¹ This means that wrap can never be used in place of a Set .

¹The exact reason for this is not well documented.

Because wrap.u is a global universe, wrap is called *universe monomorphic*. In particular, this means wrap inhabits a fixed place in the type hierarchy and any constraint applying to a particular occurrence of wrap will apply to all occurrences.

Since wrap is exactly as big as its parameter type T , we require a bit of extra setup to demonstrate the consequences of its monomorphism. We first axiomatize a monomorphic container type C which is strictly larger than its contents, as expressed by the universe constraint $C.x < C.y$. With

C , we can produce a type that is strictly larger than *some* application of `wrap`, e.g., `(wrap unit)`. This introduces the global constraint $\text{wrap.u} \leftarrow C.x$. Attempting to apply `wrap` to the resulting type would introduce the constraint $C.y \leftarrow \text{wrap.u}$, equating $C.x$ and $C.y$. This is in violation of our postulated constraint $C.x < C.y$ and thus leads to a universe inconsistency and to our construction being rejected by Rocq.

```
Axiom C@{x y | x < y} : Type@{x} -> Type@{y}.
Example ex1 := C (wrap unit).
Fail Example ex2 := wrap (ex1).
```

The example illustrates that monomorphic types only ever accumulate constraints and thus become less and less applicable with every use. This is clearly undesirable from the perspective of code re-use. To alleviate this, Rocq offers two forms of universe polymorphism, *template universe polymorphism* and *true universe polymorphism*². While template universe polymorphism predates true universe polymorphism in Rocq, it is also conceptually more complex than true universe polymorphism. We thus start our exploration of universe polymorphism with true universe polymorphism.

Our example can be adapted to true universe polymorphism with the help of the `Polymorphic` attribute and requires no further changes beyond that.

```
Polymorphic Inductive wrap (T:Type) : Type :=
| Wrap : T -> wrap T.
```

Rocq translates this declaration as follows.

```
Inductive wrap@{u} (T : Type@{u}) : Type@{u} :=
| Wrap : T -> wrap@{u} T.
```

Note the new annotation on `wrap@{u}` both in its declaration as well as in its uses. The annotations serves a dual role. In `Inductive wrap@{u}` it serves as a *universe signature* which introduces a single *universe variable* called u . In the type of the constructor `Wrap`, the annotation `wrap@{u}` designates a *universe instance* that instantiates the `wrap`'s signature with u .

Note that there is no constraint placed on u , not even $\text{Set} < u$. Unlike global universes, polymorphic universes are allowed to be equated with `Set`.

With a polymorphic declaration, there is no global universe introduced and `wrap` exists entirely outside the global universe graph. As a consequence, every occurrence of `wrap` (or its constructor) can be instantiated with different universes. To illustrate this, we revisit our previous example showcasing the restrictions placed on universe monomorphic definitions. We can re-use the code verbatim, only that this time `wrap` is universe polymorphic.

```
Axiom C@{x y | x < y} : Type@{x} -> Type@{y}.
Example ex1 := C (wrap unit).
Example ex2 := wrap (ex1). (* Succeeds! *)
```

²True universe polymorphism is usually just called “universe polymorphism” but it is useful in our context to be able to clearly distinguish the two forms of polymorphism.

To see exactly why this example now works, we can look at the fully universe annotated terms inferred by Rocq.

```
Example ex1 : Type@{C.y} := C (wrap@{Set} unit).
```

```
Example ex2 : Type@{C.y} := wrap@{C.y} ex1.
```

Universe polymorphic definitions more complex than `wrap` usually have universe constraints attached to them. These constraints become part of the type’s universe signature and are checked at application time. For example, consider the universe polymorphic dependent pair defined below with an explicit universe signature including constraints. Its return type `Type@{ur}` must be big enough to contain both components of the dependent pair, and the inductive carries the corresponding constraints in its signature.

```
Polymorphic Inductive sigma@{ut up ur | ut <= ur, up <= ur}
  (T : Type@{ut}) (P : T -> Type@{up}) : Type@{ur} :=
| Sigma : forall t : T, P t -> sigma@{ut up ur} T P.
```

We now turn to template universe polymorphism which differs from the true universe polymorphism introduced above in several important aspects. Firstly, template universe polymorphism predates true universe polymorphism and is still relied upon almost exclusively in Rocq’s standard library. In fact, template universe polymorphism is implicitly the default configuration of all (co-)inductive types in Rocq unless they are made truly universe polymorphic either through a global setting or the local Polymorphic annotation.³

Secondly, template universe polymorphism is restricted to (co-)inductive type declarations. While universe monomorphism and true universe polymorphism straightforwardly extend to ordinary `Definitions`, template universe polymorphism is tied to (co-inductive) types and their constructors. While this restriction may sound quite severe, it is usually not relevant in practice. We return to this aspect later on.

Finally, unless true universe polymorphism is actually required, template universe polymorphism is usually the better option for performance. The reasons for this are manifold and, to some degree, a consequence of the implementation details of Rocq’s universe graph algorithms. It certainly helps that `Definitions` and other non-type objects are not in any way universe polymorphic themselves even when they use template universe polymorphic types. This means all their universes are entered into the global graph which amortizes some of the universe checking work that would have to happen repeatedly if they were truly universe polymorphic.

To understand template universe polymorphism, we return to our running example. Without disabling “Auto Template Polymorphism”, our type is automatically interpreted as a template polymorphic definition

```
Inductive wrap (T:Type) : Type :=
| Wrap (t : T).
```

Again, we inspect Rocq’s interpretation of our type and notice that it looks identical to the monomorphic version. An additional query reveals

³ This behavior can be changed by the disabling the “Auto Template Polymorphism.” flag used above.

the difference to the universe monomorphic version: `wrap.u` is a *template universe* and `wrap` is thus template universe polymorphic.

```
Inductive wrap (T : Type@{wrap.u}) : Type@{wrap.u} :=
| Wrap : T -> wrap T.
About wrap.
(* [...] wrap is template universe polymorphic on wrap.u [...] *)
```

As with the monomorphic version, `wrap.u` is a global universe and exists only exactly once. Just like a monomorphic universe, it is restricted to always be strictly larger than `Set`. However, unlike with monomorphic universes, this restriction is effectively irrelevant. We will in fact be able to use the template universe polymorphic version of `wrap` at type `Set`.

The absence of a polymorphic universe signature also implies that `wrap` cannot be *instantiated* with different universe levels. Nonetheless, `wrap` can be *used* at different universe levels. The way Rocq achieves this is best illustrated by revisiting a slightly more general version of our container example. In particular, instead of starting our example with wrapping a small type such as `unit` we instead axiomatize a type `ty` at some universe `ty.u`.

```
Axiom C@{x y | x < y} : Type@{x} -> Type@{y}.
Axiom ty@{u} : Type@{u}.
Example ex1 := C (wrap ty).
Example ex2 := wrap (ex1). (* Succeeds! *)
```

This more general example succeeds with the help of template universe polymorphism (and so would the original version). The reason is that—unlike with the universe monomorphic version of `wrap` and despite what the type declaration itself suggests—applications of `wrap` do not have type `Type@{wrap.u}`. Instead, every application of `wrap` (and of its constructor) generates a copy of the template universe `wrap.u` with all its constraints. The application then follows the usual typing rules for universes only that all occurrences of `wrap.u` and its constraints are replaced by their respective copies.

In `ex1`, by virtue of being the universe of `wrap`'s argument, the fresh copy of `wrap.u` is immediately equated with the universe of `ty`'s type, `ty.u`. Following the usual typing rules for applications, the outer application of `C` incurs a constraint of the form `ty.u <= C.x`. In `ex2`, the fresh copy of `wrap.u` is equated with the universe of `ex1`'s type, `C.y`. With the template universe `wrap.u` consistently replaced by fresh copies, the two applications of `wrap` are essentially decoupled and we avoid the universe cycle we saw in the universe monomorphic version of the example.

The fact that every application creates fresh copies of the global template universe(s) also explains why the constraint `Set < wrap.u` is not a limitation in practice. Unlike `wrap.u`, the copies of `wrap.u` are allowed to be equated with `Set` as they are not themselves global universes in the global universe graph.⁴

There is one additional set of constraints generated by template universe polymorphism: all generated universe copies are always upper-

⁴ Though they may of course end up in the global universe graph or be equated with global universes, in which case they would once again have to be larger than `Set`.

bounded by their corresponding template universes. In our example above, these additional constraints amount to $\text{ty.u} \Leftarrow \text{wrap.u}$ and $\text{C.y} \Leftarrow \text{wrap.u}$ from `ex1` and `ex2`, respectively. In many cases, these additional constraints are without consequence. The reason that they are necessary is that there is one particular case in which `wrap.u` is still used directly during typechecking: when `wrap` (or its constructor) is *not applied*.⁵ Unlike its applied counterparts, the term `wrap` without arguments does actually have the type $(\text{Type@}\{\text{wrap.u}\} \rightarrow \text{Type@}\{\text{wrap.u}\})$.

⁵ For types with multiple arguments, the relevant distinction is whether they are *not fully applied*.

This corner case becomes relevant when, for example, Rocq users define type classes such as `Functor : Type@{a} -> Type@{b}` and instantiate them with template universe polymorphic types. A particularly problematic situation arises when users register a *monomorphic* type class instances on such a class for a template universe polymorphic type. Consider an implementation of `Functor`⁶ for our template universe polymorphic `wrap` type.

⁶ We ignore the laws associated with functors as they are not relevant to the universes involved.

```
Class Functor@{a b} (F : Type@{a} -> Type@{b}) := {
  fmap {X:Type@{a}} {Y:Type@{a}} : (X -> Y) -> F X -> F Y
}.
Instance wrap_functor : Functor wrap := {
  fmap _ _ f w := match w with Wrap _ x => Wrap _ (f x) end
|}.

```

Since `wrap` appears unapplied in (the type of) `wrap_functor`, Rocq needs to show that $(\text{Type@}\{\text{wrap.u}\} \rightarrow \text{Type@}\{\text{wrap.u}\})$ is a sub type of $(\text{Type@}\text{Functor.a} \rightarrow \text{Type@}\text{Functor.b})$. This incurs two constraints: $\text{wrap.u} \Leftarrow \text{Functor.b}$ and $\text{wrap.u} = \text{Functor.a}$. Both constraints are highly problematic since they upper bound the template universe `wrap.u`. All it takes now for a universe cycle is our trusted container `C` and a type that is at least as large as `Functor.a`. For simplicity, we axiomatize such a type below.

```
Axiom large_ty : Type@{Functor.a}.
Axiom C@{x y | x < y} : Type@{x} -> Type@{y}.
Example ex3 := wrap (C large_ty). (* Fails *)

```

With $\text{wrap.u} = \text{Functor.a}$, the fresh copy of `wrap.u` generated for the application of `wrap` cannot be larger than `Functor.a`, which is necessarily contained in `C.x` due to the application `C large_ty`. At the same time, the fresh universe needs to be large enough to contain `C.y` which is strictly larger than `C.x`. We thus arrive at a contradiction and the example is rejected by Rocq.

This corner case has been the reason of several incompatibilities between different Rocq libraries. Like most universe problems, universe issues arising from template universe polymorphic types are difficult to debug and have thus led to severe frustration among affected users. Thus, plans were made by Rocq developers to entirely replace template universe polymorphism by true universe polymorphism. For various reasons, chief among them backwards compatibility and performance, these plans have never been implemented beyond initial experiments. However, recent

developments indicate that there could be a way to make template polymorphism a special case of true universe polymorphism while, at the same time, fixing the existing bugs arising from unapplied template universe polymorphic types.

17.2 UNIVERSES IN MTAC2

Mtac2's core types, namely `M`, `gtactic`, `tactic`, and `ttac` are all truly universe polymorphic. While it is technically possible to use template universe polymorphism for these types, the downsides associated with un- and partially applied types pose too big a risk for Mtac2. Similarly, Mtac2 tactics are also generally (truly⁷) universe polymorphic. This is to avoid constraints accumulating in the global graph whenever the tactic is applied and executed, which would render tactics less and less useful the more they are used.

However, using universe polymorphism poses additional challenges. Firstly, there is considerable performance overhead when using polymorphic universes. Secondly, polymorphic universes by themselves are not enough to achieve the desired dynamic semantics for tactics. We go into these two issues separately below.

17.3 PERFORMANCE ISSUES WITH UNIVERSE POLYMORPHISM

There exist several factors that affect the performance of universe polymorphic Mtac2 tactics. The first factor is the sheer number of universes. One reason that leads to an unexpectedly high number of universes is that universe polymorphic definitions do not support *algebraic universes*. Algebraic universes are universe expressions that directly represent a universe level as an algebraic expression on existing levels and constants. In Rocq, this algebra includes the successor operator as well as the maximum operator, allowing terms such as $\max(\text{Type}@i, \text{Type}@j + 1)$.

The ability to express the type of a type by incrementing the universe level by 1 can significantly shrink the universes required for type-polymorphic objects in Rocq. Unfortunately, algebraic universes are only fully supported⁸ in universe monomorphic objects but not in universe polymorphic ones. Thus, while a universe monomorphic object can be declared with the type signature $\text{Type}@u \rightarrow \text{Type}@{u+1}$, a universe polymorphic object is generally forced to introduce a second universe v with $u < v$ and use the type signature $\text{Type}@u \rightarrow \text{Type}@v$.

Confounding the performance problems arising from the number of universes is the fact that Rocq generates fresh universes for every use of a universe polymorphic definition. Although Rocq has built-in heuristics to minimize and unify some of these universes, there is only so much the system can do while keeping the definitions maximally applicable.

The lack of support for algebraic universe expressions and the practical limitations of universe minimization together lead to definitions that look reasonably simple but have hundreds, sometimes thousands, of universe

⁷ We drop the *true* prefix from here on and use *universe polymorphism* to mean true universe polymorphism unless otherwise indicated.

⁸ Rocq permits itself to infer these algebraic expressions in the declaration of polymorphic inductive types but users are not permitted to write them explicitly.

levels. These universes and their constraints come with a price. The exact cost depends on the way the definitions are used.

Tactics used in other universe-polymorphic tactics will add to the number of universe parameters and constraints. This leads to a counter-productive situation where defining many small, modular tactics leads to larger and larger universe signatures and thus degraded performance. This makes typechecking tactic definitions themselves expensive, which itself can easily negate runtime performance wins that could otherwise be realized by using Mtac2.

Tactics applied in (usually monomorphic) lemmas will generate a hefty cost for checking the instantiation of the tactic's universe signature, including any constraints in the signature. Since universe signatures can grow to tremendous sizes,⁹ even just the act of creating fresh universes for a universe polymorphic tactic and importing the associated constraints into the local universe graph can add measurable overhead.

⁹ Mtac2 itself contains tactics with thousands of universes and constraints.

However, the instantiation of the tactic's universe signature with fresh universes is cheap compared to the cost of *cleaning up unused* universes. Unused universes arise from the fact that all object level terms in Mtac2 need to have types even when those types are never used in the proof term created by the tactic. To illustrate the problem, consider a contrived Mtac2 program capable of solving goals of the type `unit` by matching on the provided type.

```
Polymorphic Definition foo@{u} (T : Type@{u}) : M@{u} T :=
  mmatch T in Type@{u} as T return M@{u} T with
  | unit => M.ret tt
end.
```

Applying this program in a proof script generates a fresh universe for the universe variable `u`. However, the proof returned by the program is independent of whatever universe was generated for `u`. So the generated universe ends up unused in and will need to be collected at Qed time when Rocq performs minimization on the proof-local universe state and discards all universes that are not used in the proof term. Note, though, that some universes that are not used explicitly in the proof term may survive Qed. This is necessary, for example, when an unused universe sits strictly between two used universes, *i.e.*, $x < u < y$. Dropping the unused universe `u` would end up strengthening the theorem by allowing `x` and `y` to be only one universe level apart, instead of two. It is thus not possible to simply drop all universes that do not occur in the proof term. Instead, universe minimization needs to take them into account in order to eliminate as many redundant constraints and expose as many truly unused universes as possible.

Even with all this extra cost, universe polymorphism is still not sufficient to achieve the desired level of applicability of Mtac2 tactics. In the next section, we look at a problem that arises from the fact that universe polymorphism is a mechanism that only applies to top-level objects.

17.4 UNIVERSE-POLYMORPHIC RECURSION

Regardless of whether an Mtac2 metaprogram or tactic is universe monomorphic or universe polymorphic, at runtime its code will refer to exactly one set of universes, be they monomorphic universes or a fresh universe instance for the polymorphic universe signature. One consequence of this fact is that recursive Mtac2 programs and tactics have to share that one set of universes across all recursive calls. In short, Mtac2 lacks universe-polymorphic recursion.

To demonstrate why this is problematic, consider a tactic that generates proofs for two types of propositions: `inh_type`, whose definition is given below, and conjunctions thereof. The proposition `inh_type T` merely proves that `T` is inhabited. The proof is not extractable, allowing us to put `inh_type` into `Prop`.¹⁰

¹⁰ We could generalize `inh_type` to a universe polymorphic, propositional sigma type but this would make the example unnecessarily complex.

```
Polymorphic inh_type@{u} (T:Type{u}) := InhType (t : T).
```

To keep the example short, we also assume an oracle that produces inhabitants of arbitrary types. In real code, this would be the actual work performed by the tactic. However, it does not matter for our example how the inhabitants are produced. The axiomatized `oracle` is universe polymorphic to avoid interference with the actual universe problem we are trying to distill.

```
Polymorphic Axiom oracle : forall (T:Type), T.
```

We can now write a tactic in Ltac that constructs proofs of our propositions.¹¹ Note that `conj` is the constructor of the `and` type behind the `/\` notation.

¹¹ We construct and return a proof term instead of applying the proof term directly to the goal. This is to keep a close correspondence between the Ltac code and an Mtac2 version that will follow shortly.

```
Ltac prove c :=
  lazymatch c with
  | ?p /\ ?q =>
    let p := prove p in
    let q := prove q in
    constr:(conj p q)
  | inh_type ?a =>
    constr:(InhType a (oracle a))
  end.
```

Below, we set up a test case for our tactic that consists of a conjunction of two `inh_type` at incompatible universes.

```
Monomorphic Universe a.
Monomorphic Universe b.
Monomorphic Constraint a < b.
Axiom A : Type@{a}.
Axiom B : Type@{b}.
```

We can now ask Rocq to run the Ltac code on our test case. There is no reason for the tactic to fail and Rocq happily accepts the generated

proof. The fact that the two sides of the conjunction mention incompatible universes does not constitute a problem since there is no reason for the universes of the two sides to be related in any way.

```
Goal inh_type@{a} A /\ inh_type@{b} B.
```

```
Proof.
```

```
  let t := prove (inh_type@{a} A /\ inh_type@{b} B) in
  exact t.
```

```
Qed.
```

We now implement the same tactic in `Mtac2`. The tactic is polymorphic to avoid possible interference from monomorphic universes. In an act of blatant foreshadowing, we explicitly assign the name `u` to the universe used in the pattern that matches `inh_type` so that we can later refer to it.¹² Aside from that, the `Mtac2` implementation matches the `Ltac` code.

¹² The syntax `{u .. v+}` instructs `Rocq` to create variables for any missing universes automatically, as it usually does for all universes when no annotation is given.

```
Polymorphic Definition prove@{u+} : forall (p : Prop), M p :=
  mfix1 prove (p : Prop) : M p :=
    mmatch p as p return M p with
    | [? p q] p /\ q =>
      p <- prove p;
      q <- prove q;
      M.ret (conj p q)
    | [?a] inh_type@{u} a =>
      M.ret (InhType a (oracle a))
  end.
```

The definition itself is accepted by `Rocq`. However, if we try to run the code on our example, `Mtac2` fails with a `NoPatternMatches` exception.

```
Goal inh_type@{a} A /\ inh_type@{b} B.
```

```
Proof.
```

```
Fail mrun (prove (inh_type@{a} A /\ inh_type@{b} B)).
```

```
(* Raises a [NoPatternMatches] exception. *)
```

The exception originates in the right-hand branch of the conjunction, *i.e.*, *after* we have successfully unified our pattern `inh_type@{u}` with the left-hand side of the conjunction, `inh_type@{a}`. For this unification to succeed, `Rocq` must establish the constraint `u = a`. With this constraint, the unification of the right-hand side would need to establish `u = b`. By transitivity, this implies `a = b` and, thus, violates our ambient constraint `a < b`.

Universe-polymorphic recursion, *i.e.*, the ability to perform recursive calls at different universes, would help us avoid this particular problem. However, being able to choose different universes for recursive calls is not enough by itself. There is no fixed number of universes that we could chose upfront that would work for arbitrarily nested terms. What we require is the ability to create an unbounded number of *fresh* universes at tactic runtime to account for arbitrarily deep nesting of universe polymorphic terms.

17.4.1 A Primitive for Refreshing Universes

Instead of implementing a generic universe-polymorphic recursion primitive, we instead opt to offer a generic universe-refreshing primitive that is not directly tied to recursion. With this, Mtac2 programmers can *refresh* the universes of the recursive function before recursive calls, thus effectively mimicking universe-polymorphic recursion with the ability to generate fresh universes for the recursive call. We impose two restrictions to simplify the implementation of the primitive: (1) the primitive is only applicable to Rocq terms that carry universe annotations: top-level polymorphic entities such as definitions, (co-)fixpoints, (co-)inductives; and (2) the primitive always refreshes all the universes in the term's universe instance.

We call this new primitive `refresh_prim` and give it the type `forall (A:Type), A -> M A`. The restriction to polymorphic top-level entities can be enforced dynamically either by throwing an exception or by simply returning all other kinds of terms unchanged.

The primitive's implementation creates fresh universes for the given term and typechecks the refreshed term against its original, expected type. This will create all the necessary constraints on the fresh universes for the resulting refreshed term to be usable in place of the original term. Typechecking the refreshed term against its expected type is necessary for soundness, as Rocq's initial typechecking pass over the tactic only guarantees the well-typedness of the tactic when the original term with its original universes is used. Our replacement's universes need to be as constrained as the original for this guarantee to hold after refreshing the universes.

The restriction to polymorphic top-level entities has some consequence for clients of the primitive. In particular, we cannot apply the refreshing primitive directly to recursive occurrences of Mtac2 fixpoints. The reason is that Mtac2 fixpoints are not top-level entities¹³ and thus are not universe-polymorphic themselves, running afoul of the first restriction of this new primitive.

We thus return to our running example. However, this time we define the fixpoint in two steps to ensure that we have a top-level entity whose universes we can refresh. We keep the body of our top-level definition as close as possible to the original version using `mfix1` but delay tying the recursive knot to `prove_fix` which uses the new `refresh_prim` primitive.

```
Polymorphic Definition prove_toplevel@{u+} :=
  fun (prove: forall (p: Prop), M p) (p: Prop) =>
    mmatch p as p return M p with
    | [? p q] p /\ q =>
      p <- prove p;
      q <- prove q;
      M.ret (conj p q)
    | [?a] inh_type@{u} a =>
      M.ret (InhType a (oracle a))
    end.
```

```
Polymorphic Definition prove_fix : forall (p : Prop), M p :=
```

¹³ Recursive occurrences of an Mtac2 fixpoint only exist as binders in the higher-order argument to Mtac2's fixpoint combinators.

```

mfix1 go (p : Prop) : M p :=
  fresh <- M.refresh_prim prove_toplevel;
  fresh go p.

```

The crucial part in how `prove_fix` functions is that it never actually executes `prove_toplevel` with its original universes. Instead, it always executes a fresh copy that is generated by refreshing `prove_toplevel`. This way, the original universes assigned to `prove_toplevel` are never directly impacted by the unification happening in the actual code. With fresh universes for each side of the conjunction, our test case now passes.

```

Goal inh_type@{a} A /\ inh_type@{b} B.
Proof.
  mrun (prove_fix (inh_type@{a} A /\ inh_type@{b} B)).
Qed.

```

Performance

The approach of re-generating universes and using typechecking to insert the required universe constraints has a noticeable performance impact. In microbenchmarks that perform little to no work in each recursive call, we find a performance degradation of 10x, *i.e.*, the refreshing version of a fixpoint is 10 times slower than the non-refreshing version.

Unfortunately, unlike with many microbenchmarks, we can expect a much larger regression with code that performs more work in recursive calls. This is because more work almost always entails more universes and the algorithms involved in typechecking scale worse than linearly with the number of universes. The fact that interesting `Mtac2` programs and tactics have thousands of universes and constraints makes it infeasible to rely on fully refreshing universes at every recursive call.

17.4.2 Alternative Implementations

A smarter implementation could alleviate some of the cost of refreshing universes. The approach that seems most promising is to delay the instantiation of universe polymorphic tactics with concrete universes. In this approach, typechecking the application of a metaprogram before its execution would not create fresh universes but rather remember the constraints imposed on those universes *if* we had created them.

Then, we only actually materialize universes that *escape the interpreter*. This must happen in all primitives that interact with Rocq’s API. A prime example is unification, which can only work if all universes in the terms being unified are known to Rocq. To illustrate this, we consider the following example of a metaprogram that prints a type’s co-domain (or the type itself if it is not a product type). `print_codom` comes with explicit annotations for three universe variables, `i`, `u`, and `v`. These universe variables are used to represent the input type of the program as well as the domain and co-domain of the pattern matching branch that looks for product types.

```

Polymorphic Definition print_codom@{i u v+} :

```

```

Type@{i} -> M unit :=
mfix1 go (T : Type@{i}) : M unit :=
  mmatch T with
  | [?(X:Type@{u}) (F:X -> Type@{v})] forall X:X, F X =>
    \nu x : X, go (F x)
  | _ => M.print_term T
end.

```

Despite the fact that these universes are arguably the only relevant universes in our metaprogram, Rocq requires 12 additional universes and a total of 18 universe constraints to typecheck the metaprogram. At runtime, the only relevant interaction of this metaprogram with Rocq is through creating evvars for the first pattern matching branch, through the unification call in that branch, and through the `nu` operator. The only universes involved in those primitives are those contained in the input term `T` as well as `i`, `u`, and `v`.

To profit from this observation, `Mtac2` would typecheck the initial application of `print_codom` to its input type in a *copy* of Rocq’s original `evvarmap` (which also contains universes and universe constraints). This typechecking call will create fresh universes for `print_codom` and insert the corresponding universe constraints, *i.e.*, those imposed by the definition of `print_codom` as well as those between the fresh universe variable and the universes already contained in the input type. These universes and constraints stay in our copied `evvarmap` and remain unknown to the original `evvarmap` for now. We substitute the freshly created universes into our program and start to run it in the original `evvarmap`.

The fact that our program contains unknown universes is theoretically fine as long as we are running basic interpreter steps such as pure reductions (such as unfolding definitions hidden behind `mmatch`), `mfix1`, `M.bind`, or any other step that does not meaningfully interact with the `evvarmap`. As soon as we reach primitives that interact with the `evvarmap`, we need to carefully materialize the universes involved. Consider the creation of the domain `evvar X`. As shown in Figure 13.1, `gen_evvar` is declared as follows:

```

gen_evvar@{UA}: forall (A:Type@{UA}), M@{UA} A

```

In our code, `A` equals `Type@{u}`. Consequently, `UA` needs to be larger than `u` and Rocq does indeed generate a fresh universe with that constraint for us. Luckily, `UA` is only necessary for typechecking the metaprogram but not for creating a well-typed `evvar`. For that, we only need `A` itself, not its type. Thus, the interpreter can get away with notifying Rocq’s original `evvarmap` of the existence of `u` and *relevant constraints* associated with it. There are no constraints between `u` and any other revealed universes at this point simply because there are no other revealed universes yet. If `u` were transitively related to `Set`, we would reveal the universe and constraints involved in that transitive chain; but this is not the case in our example.

We repeat this step for the `evvar F`, revealing the universe variable `v`, any transitive constraints between `u` and `v` or `Set` and `v` as well as all universes

involved in those constraints. Finally, we reach the call to unification. The underlying primitive has the following type:

```
unify_cnt@{UA UP}: forall
  {A:Type@{UA}} {P: A->Type@{UP}} (u: Unification)
  (x y: A),
  M@{UP} (P y) -> M@{UP} (P x) -> M@{UP} (P x)
```

In our code, A is $\text{Type}@{i}$ and P is $(\text{fun } _:\text{Type}@{i} \Rightarrow \text{unit})$. Once again, UA is necessarily going to be big enough to contain both $\text{Type}@{i}$ as well as the type of P , both of which are represented through universe variables created by Rocq. Just like for `gen_evar`, these universe variables are only necessary for typechecking the metaprogram but not for the unification call itself. While unification may compute the type of its inputs, this computation depends solely on the universes present in the input terms to unification. We can once again restrict the universes we must expose to Rocq to a single one: i (including its transitive constraints with other revealed universes as well as with `Set` and all universes necessary for these constraints).

This leaves the `nu` operator which operates on the same universe u that has already been covered by the execution of `gen_evar`. We do not need to perform additional work here.

Finally, through the second branch, `print_codom` returns a value of type `unit`. `unit`, living in `Set`, has no interesting interaction with universes or universe constraints. However, we generally need to ensure that the return value of a metaprogram fits in the *expected type*. When executing a tactic on a goal of type G the expected type is either `tactic`, `ttac G`, or $M\ G$. All three types carry universes and constraints the we must consider.

One way to perform this check against the expected type is to simply typecheck the return value of the metaprogram against the expected type at the end of the execution. However, this check would already be covered if we combined our initial typechecking step of the metaprogram with a check against the expected type, making sure to expose only those universes and constraints to the original `evamap` that are required to connect the return type with the expected type.

Returning to Universe-Polymorphic Recursion.

The algorithm sketched above does not yet take into account our original goal of improving the performance of universe-polymorphic recursion. Still, it already has a significant advantage over `Mtac2`'s interpreter as it is implemented: the number of universes that are exposed to the original `evamap` is a mere fraction of the number of universes required to typecheck the metaprogram. This alone would significantly improve Qed performance (specifically universe minimization) in proofs that use `Mtac2`.

To extend the benefits of our lazy approach to revealing universes to universe-polymorphic recursion, we essentially copy what happens during the initial typechecking of the metaprogram whenever we execute

`refresh_prim`. We first refresh universes the same way we introduce universe in the first place during the initial typechecking: by computing an entirely fresh set of universes (and associated constraints) and hiding them from Rocq in our copy of the original `evarmap`. This step is the same as in our naïve universe-refreshing primitive and represents only a small part of the overall overhead.

It is the next step where we expect significant performance improvements from hiding universes: checking the refreshed term against its expected type. As with the overall metaprogram, we perform this check against the expected type in our copy of the `evarmap`, not in the original `evarmap`. Likewise, all fresh universes that are related to exposed universes through this typechecking call need to be exposed (along with associated constraints that connect them to other exposed universes). This step still incurs a significant cost that scales with the complexity of the type and the number of universes and constraints. However, unlike in our naïve attempt, most of the resulting universes and constraints will not be revealed to the original `evarmap`. This significantly reduces one compounding factor in cost of checking the refreshed type against the expected type: the number of universes and constraints in the `evarmap`.

We could even consider a more elaborate scheme that computes a first-class representation of the universe effects of checking the refreshed type against the expected type. This can be done by comparing the copied `evarmap` before and after the typechecking call with the goal to easily replay the effects with fresh universes the next time we perform the recursive call. The recorded universes and constraints can then be split into two groups: those that are transitively related to the universes in the expected type and those that are not. The latter need no further processing when they are replayed with fresh universes. The former are candidates for needing to be exposed immediately based on the status of the universes in the expected type that they are related to. If the corresponding universes in the expected type are already exposed, the fresh universes (and their associated constraints connecting them to other exposed or to-be-exposed universes) need to be exposed as well.

Evarmap bookkeeping.

Note that for supporting `refresh_prim` we need more than one copy of the original `evarmap`: as the original `evarmap` evolves, `evars`, universes, and universe constraints added to it might be necessary for the typechecking we perform in the copied `evarmap`. Thus, every call to `refresh_prim` must produce a fresh copy of the `evarmap`. Since `evarmaps` are a functional data structure, this comes at almost no additional cost. We also need to track the association of fresh universes to the copied `evarmap` in which they are registered which is easily achieved with a hash map. Neither of these requirements poses a performance concern.

Challenges.

Implementing and testing this approach remains future work. The implementation is likely far from trivial and it is unclear if Rocq's API for universes is expressive enough to implement it. It also remains to be seen if this design is indeed sound. Most importantly, this proposal needs a very careful soundness argument that includes cases for all Mtac2 primitives through which universes are exposed to Rocq.

18

Further Challenges

There are several features missing from Mtac2 that would round out its claim to being a full-fledged metaprogramming language for Rocq. In particular, Mtac2 does not officially support *Vernacular-level* metaprogramming. Rocq’s Vernacular language is the language of top-level commands. Most importantly, those commands include declarations such as `Definition` and `Inductive`. More advanced metaprogramming languages might include the ability to open modules and sections, to declare typeclass instances, and to interact with hint databases for proof automation.

A typical example that is often requested by Rocq users is the ability to automatically make a definition opaque without losing the option to selectively replace the definition by its original body. This is necessary, for example, to hide large terms from Rocq to avoid unnecessarily slow reduction, unification, or conversion calls involving those terms. This form of hiding is usually called *sealing* in Rocq. There are several ways to achieve sealing. The most straight-forward one is to use *module sealing* as is done in the example code below where we use this approach to hide the contents of a large unary natural number called `x`.

```
Module Type x_SEALED.  
  Axiom x : nat.  
  Axiom x_unfold : x = 9999.  
End x_SEALED.  
Module x_DEF : x_SEALED.  
  Definition x := 9999.  
  Definition x_unfold : x = 9999 := eq_refl.  
End x_DEF.  
Import x_DEF.
```

Running this code in Rocq yields an opaque constant `x` that can be replaced by our chosen body, `9999`, using `rewrite x_unfold`. The same result could have been achieved with a `Qed`-sealed constant that contains a dependent pair of the form `{x & x = 9999}`, using the first projection for `x` and the second projection for `x_unfold`. Either way, all approaches to sealing require additional modules or definitions and are thus prime candidates for Vernacular metaprogramming. Ideally, the metaprogram-

ming language would allow one to write a declaration that looks very similar to a version without sealing.

Another example of Vernacular metaprogramming arises in developments that rely heavily on typeclasses. Working with typeclasses often involves a fair bit of trivial instance declarations, often to the point that some Rocq developments introduce tactics to solve the proof obligations of their typeclasses so that final declaration has the following shape.

```
Instance my_def_C : C my_def.
Proof. solve_C. Qed.
```

This act of producing typeclass instances algorithmically is often called a “deriving” mechanism. Some languages such as Haskell and Rust have built-in support for deriving fundamental typeclasses such as those for comparison operators or printing together with types as they are declared. Rocq itself does not support a deriving mechanism in this style¹ but some Rocq metaprogramming languages do support arbitrary derive extensions[Tas19].

Both the act of sealing as well as the automatic derivation of typeclass instances could be done with some form of annotation of the naïve definition of `x` as is sketched below (ab)using Rocq’s notion of attributes.²

```
#[seal(module), deriving(C)]
Definition x : nat := 9999.
```

Mtac2 does not support this form of Vernacular metaprogramming. There are three reasons for this lack of support. The first reason is that the execution model of Mtac2—and, in fact, the execution model of Ltac1 and Ltac2—works perfectly for the kind of metaprogramming required in tactics but is not well suited to Vernacular-level metaprogramming. Producing Vernacular-level objects imposes restrictions on the different kinds of environments which are more stringent than the restrictions that exist when executing tactics. We go into more detail on this difference in Section 18.1.

The second reason is that Mtac2’s approach has always been “type-first”, *i.e.*, to offer strongly typed primitives that statically provide the strongest possible guarantees for whatever operation they implement. For some aspects of Vernacular-level metaprogramming, especially those related to inductive types, offering strongly typed primitives is an exercise in (partially) modeling Rocq’s theory in Rocq itself. Section 18.2 sketches what it would take to design these strongly-typed metaprogramming primitives.

The last and rather pragmatic reason is that Mtac2 development was largely focused on tactic programming, which to a large degree requires only the subset of metaprogramming that deals with term manipulation. There simply was little overlap with Vernacular-level metaprogramming.

There is one additional aspect that would likely hamper adoption of Mtac2: There is no support for unsafe term inspection and construction. This feature is relevant for performance in cases where tactics construct big terms piece-by-piece. Checks such as typechecks or even closedness

¹ There is a `Derive Vernacular` command that does allow defining a term together with a proof of some property of it but it is limited to a single property and it does not have support for automatically registering the proof as a typeclass instance. Both of these restrictions severely limit the potential reduction in boilerplate.

[Tas19] Tassi, “Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq”

² Due to technical limitations of attributes this sketch is a straw man proposal that cannot be implemented exactly as is.

checks could often be done more efficiently if they are only performed on the fully assembled term instead of being repeated for ever-growing subterms. Additionally, it is sometimes too arduous to guarantee the well-typedness of every subterm. Even worse, it is sometimes simply not possible to construct the term given Mtac2's limited set of primitives. For these reasons, an unsafe escape hatch in Mtac2 that supports basic AST-based term inspect and construction would be a great extension. A simple solution is outlined in Section 18.3.

18.1 META- AND TACTIC PROGRAMMING: NOT ENTIRELY THE SAME

It is generally true that tactic programming in Rocq is meta programming: tactics are programs that inspect and generate programs (proof terms). Unfortunately, the properties that make tactic languages convenient to work with also make them particularly bad platforms for Vernacular-level metaprogramming tasks.

The problem comes from the implicit, ambient state that is present at tactic execution time. Tactic languages in Rocq that want to interact with Rocq's own notion of goals and proof scripts need to be programmed against an interface called *Proofview* which offers a monadic encapsulation of the proof state. Inherent to this representation is the concept of ambient environments and *evarmaps*, the latter containing information about existential variables and universe variables that were introduced in the course of the proof. While environments are manipulated in goal-local ways, the same is not true for *evarmaps*. Existential variables as well as universes shared between goals introduce non-locality that makes it impossible to manipulate individual goals independently³.

This interaction with environments and *evarmaps* is very different to the conditions at Rocq's top level where, for example, new definitions are introduced. At definition time, the constituents of the new definition need to be entirely free of *evvars* and closed in the global (top-level) environment. Rocq offers a layer of convenience around this bare-bones notion of a definition by allowing holes to appear in the terms and types that can then be inferred by the *pretyper*. And while these holes are existential variables just like those that are created in tactics, the corresponding *evarmap* that contains them is transient and disappears as soon as the elaborated term is given to the kernel.

In other words, tactics operate on an ever-growing and shared *evarmap*, whereas metaprograms generally operate without an ambient *evarmap* but get access to transient *evarmaps* that exist only for the duration of pretyping the constituent terms of the top-level items they create. The more a tactic language relies on the ambient *evarmap*, the more difficult it is going to be to integrate Vernacular-level metaprogramming into the language.

How much this distinction matters depends entirely on the degree that typical programs in any given metaprogram depend on the ambient *evarmap*. A metaprogramming language that deals entirely in ASTs

³ Connecting different goals through shared state is the whole reason for the existence of *evarmaps*.

will likely be able to integrate support for Vernacular-level metaprogramming with relative ease. `Mtac2`, at the other end of the spectrum, depends crucially on the presence of an ambient `evarmap` for its type safety guarantees—which cannot even be spelled out without referring to the ambient state. After all, the `M` monad of `Mtac2` is, at the very basic level, a state monad that encapsulate the `evarmap` and some additional environments.

This poses a fundamental restriction on how support for Vernacular-level metaprogramming could be integrated into `Mtac2`. Specifically, this means that if `Mtac2` were to be extended to support the primitives that can declare definitions and other entities, one would have to carefully categorize all primitives according to whether they require an ambient `evarmap` to function. Those that do would have to be made type-distinct from those that do not, thereby carving out a *pure fragment* of the language that can be executed without an ambient `evarmap`.

The least invasive (but still quite invasive) way to perform such a split is by defining a new monad `V` (for “Vernacular”), that is a *subtype* of the `M` monad. One way to achieve this subtyping relation is by using universes⁴. We introduce a `protoM` monad that is parametrized by one universe `t` for its type argument and also one additional, new universe `p` that we will use to distinguish Vernacular-level from the remaining primitives. We then introduce two incompatible global monomorphic universes, `vernac` and `generic` that we use to define `V` and `M`, respectively.

⁴ Subtyping through universes provides a very clear presentation of splitting the `M` monad. Compared to coercions, it also has the advantage of leaving no additional terms at elaboration time. However, given the trouble `Mtac2` has with the number of universes it seems unwise to introduce even more of them. An actual implementation of the split might have to rely on alternative subtyping mechanisms.

```

protoM@{+p t} : forall (T : Type@{t}), Prop.
Monomorphic Universe generic.
Monomorphic Universe vernac.
Monomorphic Constraint vernac < generic.
V@{t} : forall (T : Type@{t}), Prop := protoM@{vernac t}.
ret@{UA}:
  forall {A:Type@{UA}}, A -> V@{UA} A
bind@{UA UB}:
  forall {A:Type@{UA}} {B:Type@{UB}},
    V@{UA} A -> (A -> V@{UB} B) -> V@{UB} B.
(* .. and all other primitives that do not introduce
   evars or universes/universe constraints. *)
M@{t} : forall (T : Type@{t}), Prop := protoM@{generic t}.
(* All remaining primitives use the \texttt{M} monad. *)

```

⁵ The prefix `+` annotation on the universe `p` indicates that `protoM` is covariant in `p`. It is not the same as the stand-alone `+` operator which instructs `Rocq` to add universe variables for all missing universes.

[TS18] Timany et al., “Cumulative Inductive Types in Coq”

As long as `protoM` is cumulatively⁵ universe polymorphic[TS18], this setup allows us to use terms of type `V T` in place of `M T`, *i.e.*, to use Vernacular-level code in Vernacular-level *and* generic contexts. Embedding generic computation in Vernacular-level computation can be done via a primitive that creates a temporary `evarmap`, runs the generic code, and then checks that result is valid in the original `evarmap`. This operation is similar to how the `pretyper` is called to resolve holes in `Rocq`’s own Vernacular declaration commands (*e.g.*, `Definition`) before the final, `evar-closed` term is given to the kernel.

```

in_evmap@{t} (T : Type@{t}): M@{t} T -> P@{t} T.

```

The `in_evarmap` primitive would only be allowed to return `evvar`-closed values and this condition would be enforced dynamically. The resulting values could then be given to primitives that perform Vernacular-level scripting such as declaring definitions, inductive types, etc. Those Vernacular-level primitives are all defined in the `V` monad. Finally, the existing Vernacular command `Mtac Do`, which currently executes its metaprogram/tactic argument in a temporary goal, would instead be restricted to programs of type `V unit`.

Beyond evvars: `nu` in Vernacular-level metaprogramming.

The text above focuses on the `evvarmap` which tracks `evvars`, universe and universe constraints. There is an additional environment that must always be empty when running Vernacular-level metaprograms: the local context. For a first prototype, it would make sense implement this restriction literally, *i.e.*, by keeping the `nu`-primitive in the `M` monad. However, given that `nu` is one of the most important primitives of `Mtac2`, one quality of life improvement would be to allow non-empty contexts to stand in for parameters of Vernacular-level objects created in metaprograms. This could substantially simplify the declaration of objects with complex types.

18.2 MODELING ROCQ IN ROCQ, PARTIALLY

The previous section sketches how to extend the operational model of `Mtac2` to support Vernacular-level metaprogramming. What is missing is a design that extends `Mtac2`'s tradition of strongly typed primitives to the interesting aspects of Vernacular-level metaprogramming. To illustrate the problem, consider a hypothetical primitive that declares a definition.⁶

```
declare_def {A:Type} (name : string) (a : A), M unit.
```

It may be a slightly reductive description of what constitutes a definition in `Rocq` but it is arguably true that one needs at least a name, a type, and a value of that type. Ignoring questions of the validity of `name` as a `Rocq` identifier, calls to `declare_def` could be expected to always succeed assuming the metaprogram that executes the primitive itself is well-typed and the provided values are closed in the global environment.

Now, consider instead a primitive that declares an inductive type. Is there any hope of capturing its preconditions in a similarly succinct manner as we did for `declare_def`? The answer to this question is very clearly “No”. However, this has not stopped the author from trying to achieve the strongest type guarantees that are possible.⁷ The benefits are two-fold. 1) At the time that this project started (2019-2020), `Rocq`'s API for declaring Vernacular-level objects was difficult to use, lacking documentation about the invariants assumed on the terms and types provided by callers. 2) Writing error handling code is very time consuming and tedious. If an `Mtac2` primitive can offload any failure cases to the typechecker this is often the most expeditious way of implementing it.

Unfortunately, the project ran into many practical limitations of `Mtac2` and `Rocq` itself. First and foremost, the fact that universe polymorphism is

⁶ We set aside the `V` monad for the remainder of this section to simplify the presentation.

⁷ I can only recommend this exercise to anybody who does not need to finish a PhD. It is a great way to learn about type theory and an equally great way to lose all hope of making any kind of tangible progress.

tied to top-level objects makes it all but impossible to provide reasonable primitives that declare universe-polymorphic top-level objects, such as definitions and inductives. Unlike, say, parameters of an inductive type—which can easily be modeled using function types—there is simply no way model universe signatures in Rocq itself.

The best that Mtac2 can do in this situation is to provide Rocq with a term that contains universes generated when typechecking the metaprogram that generates the term. Assuming these universes are all polymorphic universe variables (which will be the case if the metaprogram is universe polymorphic), Rocq will then abstract the universe variables used in the term (and constraints between them) and turn them into a universe signature. This can work for simple examples but it is extremely difficult to estimate what the resulting signature will look like.

The following example illustrates the problem. We consider a slightly extended `declare_def` primitive that takes an additional argument to indicate if the resulting definition should be universe monomorphic or universe polymorphic. First, we define the expected result of our `declare_def` call.

Polymorphic

```
Definition foo@{u1 u2 u3 +| u1 < u2, u2 < u3 +} : Type :=
  Type@{u1} -> Type@{u3}.
```

The term `foo` is simply a shorthand for the implication type with the extra constraint that the domain and co-domain must at least one universe level apart. While this example is contrived, it is something that could conceivably be defined by a user.

If we try to construct a call to our hypothetical `declare_def` primitive that produces `foo`, we immediately run into trouble: without a top-level object, there is no way to introduce universe variables and universe constraints. Without a top-level object, the closest we get is to define a shorthand for `Type -> Type`, which falls short of our original target due to the missing constraint between the types' universe levels.

```
Mtac Do declare_def Poly "foo" (Type -> Type).
About foo.
(*
foo@{u1 u3 ut} : Type@{ut}
(* u1 u3 ut |= u1 < ut
      u3 < ut *)
*)
```

If we want to reproduce the exact universes and constraints, our only option is to encapsulate the metaprogram in its own `Definition` with exactly the required universes and constraints.

Polymorphic

```
Definition mk_foo@{u1 u2 u3+| u1 < u2, u2 < u3} : M unit :=
  declare_def Poly "foo" (Type@{u1} -> Type@{u3}).
Mtac Do mk_foo.
```

This, of course, defeats the entire point of Vernacular-level metaprogramming.

`foo` demonstrates what happens when users want to introduce additional universes and universe constraints (on top of what Rocq would infer itself). There also exists the the dual issue that is equally difficult to solve which is when users want fine-grained control over exactly which universes and constraints go into their definitions. Both use cases would need to be addressed if Mtac2 wanted to offer general Vernacular-level metaprogramming.

18.2.1 Sort Polymorphism

Setting aside the issue with universes, there is also the challenge of supporting inductives in different *sorts*. Using `Type` everywhere and relying on Rocq implicitly lifting other sorts to `Type` does not work for declaring inductive types which need to announce their target sort upfront and which must adhere to exactly that sort in their entire declaration. While offering distinct primitives for declaring inductives in `Prop`, `Type`, and the recently added `SProp` sort is a possibility, it also vastly increases the amount of code that needs to be kept correct and in sync.

The `Sort` type introduced in Section 14.2 can be used to implement the metaprogramming primitives in a sort polymorphic way. Regrettably, it also adds a significant amount of complexity debt to the already strained complexity budget of Mtac2, both in terms of the user interface for these primitives (which can sometimes be addressed by introducing specialized version for each `Sort`) as well as term size and universes.

18.2.2 Approximately Modeling Inductive Types

Even at the more fundamental level of the *shapes of terms* that go into an inductive type, there is an astounding amount of complexity to be tackled. The model sketched below is inspired by the concept of inductive descriptions introduced in [Mcb10].

[Mcb10] McBride, “Ornamental algebras, algebraic ornaments”

An inductive type in Rocq consists of an *inductive arity*, i.e., a functional type ending in a sort, whose binders may well be dependent. This introduces a first layer of telescopes in the model of inductive types. To distinguish parameters from indices, the arity telescope is further split into one parameter telescope and an index telescope that may depend on the parameters.

To sketch these types, we first introduce telescopes and some required telescope constructions. We assume a polymorphic type of dependent pairs, `sigT` with its constructor `existT`, and use the standard Rocq syntax `{ x : X & F x }` for the type. To simplify the code, we ignore sort polymorphism in our telescope constructions.

```
Polymorphic Inductive tele : Type :=
| Tele0 : tele
| TeleS {X:Type} : (X -> tele) -> tele.
Polymorphic Fixpoint tele_fun (t : tele) (R : Type) : Type :=
  match t with
```

```

| Tele0 => R
| TeleS X t => forall x:X, tele_fun (t x) R
end.
Polymorphic Fixpoint tele_args (t : tele) : Type :=
  match t with
  | Tele0 => unit
  | TeleS X t => { x : X & tele_args (t x) }
  end.
Polymorphic Fixpoint tele_app {t : tele} {R : Type} :
  tele_fun t R -> tele_args t -> R :=
  match t as t return tele_fun t R -> tele_args t -> R with
  | Tele0 => fun (r : R) (_ : unit) => r
  | TeleS X t =>
    fun (f : forall (x:X), tele_fun _ R) '(existT x args) =>
      tele_app (f x) args
  end.
Polymorphic Fixpoint tele_bind {t : tele} {R : Type} :
  (tele_args t -> R) -> tele_fun t R :=
  match t as t return (tele_args t -> R) -> tele_fun t R with
  | Tele0 => fun (f : unit -> R) => f tt
  | TeleS X t =>
    fun (f : { x : X & tele_args (t x) } -> R) (x : X) =>
      tele_bind (fun (args : tele_args (t x)) =>
        f (existT _ x args)
      )
  end.
end.

```

With these constructions we can approximate the type of arities of inductive types and do so below. We include the inductive's sort using the Sort type from Section 14.2.

```

Polymorphic Definition inds_tele (params : tele) : Type :=
  tele_fun params tele.
Polymorphic Record arity : Type :=
{ params : tele;
  inds : inds_tele params;
  sort : Sort }.

```

To illustrate how arity works, consider Rocq's standard eq type. Its Rocq definition is given below together with a suitable arity for it. To cover recursive inductive types, we also include the type of length indexed lists, vec, and its arity.

```

Inductive eq {A:Type} (a : A) : A -> Prop :=
| eq_refl : eq a a.
Definition eq_ar : arity :=
{| params :=
  TeleS (fun (A:Type) => TeleS (fun (a:A) => Tele0))
; inds (A:Type) (a:A) := TeleS (fun _:A => Tele0)
; sort := S_Prop |}.

```

```

Inductive vec (A : Type) : nat -> Type :=
| vnil : vec A 0
| vcons {n} : A -> vec A n -> vec A (S n).
Definition vec_ar : arity :=
{| params := TeleS (fun A:Type => Tele0)
; inds (A:Type) := TeleS (fun n:nat => Tele0)
; sort := S_Type |}.

```

Note that this model is too lax compared to the true requirements Rocq imposes on inductive types. First and foremost, in our model, the telescope of indices is fully dependent in the (telescope of) parameters. This is not allowed in Rocq. Consider, for example, the following arity definable in our model.

```

Definition forbidden_arity : arity :=
{| params := TeleS (fun (b : bool) => Tele0)
; inds := fun (b : bool) =>
    if b then
        Tele0
    else
        (TeleS (fun (x : nat) => Tele0))
; sort := S_Type |}.

```

The inductive type corresponding to `forbidden_arity` is sketched below. Declarations of this shape are rejected by Rocq.

```

Inductive ... (b : bool) :
  if b then Type else forall (x : nat), Type :=
| ... .

```

In fact, this undesired flexibility in the model affects not only cross-telescope dependencies such as those between inductive parameters and indices but also dependencies within each telescope. Given that our telescopes are modeled using function spaces, each telescope segment is allowed to change the shape of the telescope's continuation based on the value of preceding binders. There are models of telescopes, including those indexed by their depth, that rule out such illegal constructions. In practice, the benefit of restricting the model further is likely marginal compared to the typing guarantees that the model already provides over an untyped and unsafe approach to declaring inductive types. It is thus possible that the benefits do not outweigh the downsides of even more complex telescope constructions. However, the optimal point in this trade-off still needs to be determined and that process would ideally involve input from users with different use cases.

We return to our attempt at modeling inductive types by turning our attention to constructors. Constructors also have possibly dependent binders which also require telescopes to model. But those arguments are allowed to mention the inductive type itself, *i.e.*, they are allowed to contain recursive occurrences of the type. Thus, constructor arguments should be modeled as functions from a value of the type's signature to

a telescope. At the same time, constructors must construct values for all the type's indices using the values of their own arguments. This can be modeled by a function from values of the constructor's telescope to values of the type's indices. Finally, the entire constructor is defined in an ambient environment that contains values of the inductive type's parameters.⁸

⁸ Note that Rocq allows recursive occurrences of the inductive type with parameter values that differ from the ambient parameter values.

Polymorphic Definition `rec_occ`

```
(S : Sort)
(params : tele)
(inds : inds_tele params): Type :=
forall (ps' : tele_args params),
  tele_fun (tele_app inds ps') (stype_of S).
```

Polymorphic Definition `constructor`

```
(S : Sort)
(params : tele)
(inds : inds_tele params) : Type :=
forall
  (self : rec_occ S params inds) (* rec. occurrences *)
  (ps : tele_args params),
{ args : tele & (* arguments of the constructor *)
  (* the indices chosen by the constructor *)
  tele_fun args (tele_args (tele_app inds ps))
}.
```

With these definitions, we can describe the constructors of `eq` and `vec`.

Definition `eq_refl_constructor` :

```
constructor S_Type (eq_ar.(params)) (eq_ar.(inds)) :=
fun _eq '(existT A (existT a _)) =>
  existT _ Tele0 (existT _ a tt).
```

Definition `vnil_constructor` :

```
constructor S_Type (vec_ar.(params)) (vec_ar.(inds)) :=
fun vec '(existT A _) =>
  existT _ Tele0 (existT _ 0 tt).
```

Definition `vcons_constructor` :

```
constructor S_Type (vec_ar.(params)) (vec_ar.(inds)) :=
fun vec '(existT A _) =>
  existT _
    (TeleS (fun (n:nat) =>
      TeleS (fun (a:A) =>
        TeleS (fun (v:vec (existT _ A tt) n) =>
          Tele0
        ))))
    (fun n a v => existT _ (S n) tt).
```

Another requirement on inductive types that is not automatically enforced by this model is that of strict positivity of recursive occurrences in

the types of constructors. There are several ways this could be addressed, even without sacrificing the convenient HOAS-inspired approach we use for recursive occurrences. A light-weight (but not bulletproof) solution could be to encode strict positivity into a typeclass and use that to provide a *safe* wrapper around the *unsafe* primitive for defining inductive types.

Our simplistic model also ignores the existence of mutually-inductive types. Instead of a single inductive type, we would need to consider a bundle of inductive types and each type’s constructors could mention its corresponding inductive type as well as all other types in the bundle. Inductive types in the bundle can, of course, have different inductive arities.

The model also deviates from the prototype implemented in Mtac2 in the exact shape used to represent recursive occurrences: the code presented above uses bundled parameters (represented using nested sigma types) and thus produces a term that requires post-processing before it can be given to Rocq. In `vcons_constructor`, we produce `vec (existT _ A tt) n` which must be turned into `vec A n` before it is given to Rocq. This change was made to simplify the presentation. In the prototype code, everything is set up such that the arguments are exactly in the shape that Rocq’s kernel expects. However, this requires additional telescope constructions which only distract from the basic ideas outlined here.

While Mtac2 already provides a prototype implementation of this model and the corresponding primitives that allow inspecting existing and declaring new inductive types, the unfortunate reality is that the many telescopes involved in a model such as this bring Rocq to its limits. Programs that handle these telescopes abstractly can often still be defined. Similarly, providing values of these types from OCaml, for example in a primitive that produces a description of an existing type, is also still feasible—although it involves a lot of careful unsafe construction of heavily universe-polymorphic terms. However, attempting to construct any non-trivial concrete instance of such a model in Rocq itself runs afoul of several performance issues in Rocq, such as one unsolved (albeit slightly improved) issue in the typechecker[Kai20c].

The terms are also so complex that it becomes extremely difficult to trace what is happening if Rocq fails to typecheck them. Typechecking failures in extremely dependent code such as the model outlined above happen very frequently. One major reason is uni-directional nature of Rocq’s typechecking algorithm. Bi-directional typechecking annotations help to limit the number of spurious failures but they also introduce extreme performance regressions in typechecking time[Kai20b].

Due to the highly universe-polymorphic nature of the model, it is also very difficult to understand universe constraint failures. Here, too, Rocq offered options that helped reduce the rate of constraint failures seen during the work on this project in the form of *cumulative universe polymorphism*[TS18]. Without enabling cumulativity explicitly for custom types, Rocq’s subtyping is limited to built-in sorts, *i.e.*, $\text{Prop} \leq \text{Type}@{\dots}$, and $\text{Type}@{i} \leq \text{Type}@{j}$ if $i \leq j$. Polymorphic inductive types, by default, do not benefit from subtyping. For example, a list of the universe poly-

[Kai20c] Kaiser, “Type checking fails very slowly. #11488”

[Kai20b] Kaiser, “Enabling bidirectionality leads to extreme slowdown. #11598”

[TS18] Timany et al., “Cumulative Inductive Types in Coq”

morphic type `list@{i}` could not be used in place of `list@{j}` even if $i \leq j$. Cumulative universe polymorphism addresses this issue and allows some terms to typecheck that do not typecheck without it.

[Kai20a] Kaiser, “Cumulativity checking(?) extremely slow #11741”

Enabling cumulative universe polymorphism did initially trigger catastrophic performance regressions[Kai20a]. Fortunately, this bug has since been fixed. However, it still remains to be seen if cumulativity is indeed required in this particular application or merely helpful in reducing the occurrence of other universe-related bugs.

It is fair to summarize the result of this project as a failure. Providing a strongly-typed API for declaring top-level objects such as inductive types is not something that Rocq itself is able to handle at this time. Even if this were to change, it is entirely unclear if human clients of such APIs could realistically understand and use these APIs. In the end, their use might always be limited to abstract libraries (likely provided by Mtac2 itself) that perform general transformations on existing inductive types and other top-level objects. Clients that want to deal with more concrete inductive types might prefer unsafe, untyped APIs that perform all their checks at runtime.

18.3 MISSING SUPPORT FOR UNSAFE OPERATIONS

The design philosophy of Mtac1 and Mtac2 has always been to provide a strongly-typed meta-programming and tactic language whose type system is the type system of Rocq. This approach has enormous benefits such as instant feedback due to informative typechecking at definition time as well as performance benefits from (generally) not having to re-check types during the execution of meta-programs and tactics.

However, as we have seen in the previous section, there is a flip-side to this insistence on strong well-typedness: Sometimes, ensuring that Mtac2 code is well-typed is prohibitive both in terms of development time (building typed representations of complex Rocq terms such as inductive types) as well as execution time (dynamic checks to uphold well-typedness invariants). It thus seems desirable to introduce a notion of unsafe operations.

One straightforward way of integrating unsafe operations is to offer a weakly-typed API for term manipulation in which terms are represented by a Gallina AST that follows Rocq’s internal representation of terms at the OCaml level. Two additional operations are needed to make this representation useful:

```
kind : forall {T: Type}, T -> M term
make : forall (T: Type), term -> M T
```

The `kind` primitive cannot fail but nonetheless must be monadic since it inspects the syntax of the given term. `make` can fail if the given term cannot be checked against the given type `T`.

There are countless additional utility functions that would improve the usability of this particular approach (many of them dealing with

substitutions). Fortunately, with `term` being a Gallina type, all these functions can in fact be implemented in Gallina itself. Rocq’s `vm_compute` and `native_compute` mechanisms can then be used to perform these operations efficiently.

This approach to term construction is completely standard and it is maximally expressive in that it allows users to construct any term they wish. The MetaRocq⁹ project[Soz+20] already provides Gallina types that represent Rocq terms and utility functions on those types as part of a formalization of Rocq in Rocq.

Apart from re-using these definitions in `Mtac2` to represent the syntax of Rocq terms, MetaRocq could also be used to skip typechecking in `make` by providing proofs of well-typedness. As a very first step, `Mtac2` could provide a primitive that takes a term representing a type and a term representing an inhabitant of that type together with MetaRocq proofs certifying their well-formedness and the relation between them. Using MetaRocq’s typing relation for the types of these proofs, the primitive could have the following signature¹⁰:

```
make_cert_sigma : forall (T v : term),
  (forall \Sigma \Gamma (s : sort),
    typing \Sigma \Gamma T (tSort s) /\
    typing \Sigma \Gamma v T
  ) ->
  M { T : Type & T }.
```

`make_cert_sigma` is not very useful on its own since the type of the returned value is existentially qualified. Despite this limitation, it could already be useful in Vernacular-level which declare top-level objects constructed unsafely and have no further need to remember or re-use the types of these objects. It remains to be seen if the connection between `Mtac2` and MetaRocq could be tightened in such a way that `Mtac2`’s tracking of the expected return type (as is done by `make` and also all the standard primitives of `Mtac2`) is not entirely lost. The key challenge is to find a way to relate types (and terms) with their MetaRocq representation in such a way that the (type of the) result of `make_cert_sigma` can be known statically.

⁹ MetaRocq was previously called MetaCoq.

[Soz+20] Sozeau et al., “The MetaCoq Project”

¹⁰ To keep the example simple we ignore difference between sorts and always return a `Type`. This could be fixed by using `Mtac2`’s sort polymorphism library.

Bibliography

- [GKY22] Gaëtan Gilbert, Jan-Oliver Kaiser, and Joshua Yanovski. “Delay Universe Substitution in CClosure #15807”. 2022. URL: <https://github.com/coq/coq/pull/15807> (visited on 06/01/2024) (cit. on p. 201).
- [Kai20a] Jan-Oliver Kaiser. “Cumulativity checking(?) extremely slow #11741”. 2020. URL: <https://github.com/coq/coq/issues/11741> (visited on 06/01/2024) (cit. on p. 230).
- [Kai20b] Jan-Oliver Kaiser. “Enabling bidirectionality leads to extreme slowdown. #11598”. 2020. URL: <https://github.com/coq/coq/issues/11598> (visited on 06/01/2024) (cit. on p. 229).
- [Kai20c] Jan-Oliver Kaiser. “Type checking fails very slowly. #11488”. 2020. URL: <https://github.com/coq/coq/issues/11488> (visited on 06/01/2024) (cit. on p. 229).
- [Mcb10] Conor McBride. “Ornamental algebras, algebraic ornaments”. Jan. 2010 (cit. on p. 225).
- [Soz+20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. “The MetaCoq Project”. In: *J. Autom. Reason.* 64.5 (2020), pp. 947–999 (cit. on p. 231).
- [Tas19] Enrico Tassi. “Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq”. In: *ITP*. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 29:1–29:18 (cit. on p. 220).
- [TS18] Amin Timany and Matthieu Sozeau. “Cumulative Inductive Types in Coq”. In: *FSCD 2018 - 3rd International Conference on Formal Structures for Computation and Deduction*. Oxford, United Kingdom, July 2018. DOI: [10.4230/LIPIcs.FSCD.2018.29](https://doi.org/10.4230/LIPIcs.FSCD.2018.29). URL: <https://inria.hal.science/hal-01952037> (cit. on pp. 222, 229).