Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq

SIMON SPIES, MPI-SWS, Germany
LENNARD GÄHER, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

Over the past two decades, there has been a great deal of progress on verification of full functional correctness of programs using separation logic, sometimes even producing "foundational" proofs in proof assistants like Coq. Unfortunately, even though existing approaches to this problem provide significant support for automated verification, they still incur a significant *specification overhead*: the user must supply the specification against which the program is verified, and the specification may be long, complex, or tedious to formulate.

In this paper, we introduce Quiver, the first technique for *inferring* functional correctness specifications in separation logic while simultaneously verifying foundationally that they are correct. To guide Quiver towards the final specification, we take hints from the user in the form of *a specification sketch*, and then complete the sketch using inference. To do so, Quiver introduces a new *abductive deductive verification* technique, which integrates ideas from abductive inference (for specification inference) together with deductive separation logic automation (for foundational verification). The result is that users have to provide some guidance, but significantly less than with traditional deductive verification techniques based on separation logic. We have evaluated Quiver on a range of case studies, including code from popular open-source libraries.

CCS Concepts: • **Theory of computation → Separation logic**; **Program specifications**; **Program verification**.

Additional Key Words and Phrases: specification inference, abduction, functional correctness, Iris, Coq

ACM Reference Format:
Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. 2024. Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq. *Proc. ACM Program. Lang.* 8, PLDI, Article 183 (June 2024), 25 pages. https://doi.org/10.1145/3656413

1 INTRODUCTION

The problem of how to verify *functional correctness* of large, stateful programs is one of the oldest challenges of computer science, tracing back to the work of Hoare [24] and Floyd [21]. Over the past two decades, remarkable progress has been made following the advent of *separation logic* [41, 49], an extension of Hoare logic that supports modular reasoning about stateful resources. Based on the foundation of separation logic, a number of powerful *deductive verification tools* have been built, including VeriFast [27], CFML [11], Bedrock [12], GRASShopper [43], VST [2, 7], Viper [39], Gillian [53, 36], Perennial [8], RefinedC [52], and CN [45]. They provide exceptionally strong verification guarantees (*e.g.*, memory safety and functional correctness in a pointer-manipulating

Authors' addresses: Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

ACM 2475-1421/2024/6-ART183
https://doi.org/10.1145/3656413

language like C) by working with rich forms of separation logic, featuring, *e.g.*, custom predicates, user-defined functions, detailed programming language semantics, and large mathematical theories. In many cases, they even establish these guarantees *foundationally*—by producing machine checked proofs in proof assistants like Coq [13].

Sadly, the overhead of functional correctness verification is considerable. For example, for manual verification in a proof assistant, the proofs are typically an order of magnitude larger than the code (see, *e.g.*, [9, Fig. 12] and [7, §11]). Hence, one of the longstanding goals has been to reduce this overhead—to scale verification to larger code bases, lower the entry barrier, and reduce the overall effort. Toward this goal, many of the aforementioned techniques have made great strides by developing *proof automation* (*e.g.*, [27, 43, 39, 52, 53, 45]), which often shrinks the overhead to—or even below—the code size (see, *e.g.*, [45, §5]). Instead of manual correctness proofs, these tools take as input the program code and one specification per function and then validate the program against the specification. To be precise, they offer the verification paradigm:

$$\text{code} + \text{specification} \; (+ \text{annotations} \; \& \; \text{proofs}) \Rightarrow \text{yes/no} \qquad \textit{(program verification)}$$

where the user provides *code*, *specification*, and in some cases additional *annotations* (*e.g.*, to guide the proof search) and *proofs* (*e.g.*, lemmas in a proof assistant) and, then, the tool outputs *yes* (possibly with a foundational proof) or *no* (with an error message).

However, for verification of functional correctness to truly scale one day, we believe that proof automation alone is not enough. The reason is that, in the current verification paradigm, even if the number of *annotations* decreases in the future and the *proof* component shrinks due to better proof automation, the resulting tools still require their users to *provide specifications manually*. And these specifications are rarely small. For a recent example, take RefinedC [52]. RefinedC targets proof automation for foundational verification of C code. Correspondingly, regarding *proof overhead*, its "relative annotation overhead is moderate—less than 0.7 for all examples that do not involve complex side conditions" [52, §7]. But this statistic does not include the *specification overhead*, which is considerable in its own right: specifications contribute an additional 30-50 percent of the code size to the total verification overhead [52, Fig. 7]. Moreover, specifications often impose as much of a burden on the verification effort as do annotations, forcing the user to supply tedious side conditions about integer arithmetic, nontrivial preconditions about pointers, error cases, and conditionals over the possible return values.

The next frontier: specification inference. So how can we meaningfully reduce the *specification overhead* of deductive separation logic techniques? Our answer to this question is to fundamentally change the verification paradigm. Instead of treating the specification as an *input*, make it an *output* of the verification:

$$\text{code} \; (+ \text{specification sketches}) \Rightarrow \text{specification/failure} \qquad \textit{(specification inference)}$$

That is, given the code together with possibly additional hints in the form of specification sketches (explained below), our goal is to infer a specification based on the code. Working in separation logic, this specification can then be used (1) compositionally in the verification of other code, (2) to infer specifications of clients, or (3) by humans to compare the specification against their expectations.

Specification inference is an even harder problem than traditional program verification—it decreases the user-provided input and increases the desired output. Accordingly, it is not solved all at once but requires a longer journey. With **Quiver**, we embark on the crucial next steps of this journey. Quiver is the first technique for inferring—and foundationally verifying—functional correctness specifications in separation logic. It takes in sketches of function specifications and completes them to a full separation logic specification—adding missing preconditions, inferring postconditions, and filling out user-determined holes. To achieve this goal, Quiver proposes a

$$\{n \in \texttt{size\_t}\}\ \mathrm{malloc}(n)\ \{v.\ v = \mathsf{NULL} \lor (\exists \ell, w.\ v = \ell * \ell \mapsto w * \mathrm{uninit}(w, n) * \mathrm{block}\,\ell\,n)\}$$

$$\{\ell \mapsto v * \mathrm{uninit}(v, n) * n \in \texttt{size\_t}\}\ \mathrm{memset}(\ell, 0, n)\ \{\_.\ \exists w.\ \ell \mapsto w * \mathrm{zeros}(w, n)\}$$

$$\{\mathsf{True}\}\ \mathrm{abort}()\ \{\_.\ \mathsf{False}\} \quad \{n \in \texttt{size\_t}\}\ \mathrm{xmalloc}(n)\ \{v.\ \exists \ell, w.\ v = \ell * \ell \mapsto w * \mathrm{uninit}(w, n) * \mathrm{block}\,\ell\,n\}$$

$$\{n \in \texttt{size\_t}\}\ \mathrm{xzalloc}(n)\ \{v.\ \exists \ell, w.\ v = \ell * \ell \mapsto w * \mathrm{zeros}(w, n) * \mathrm{block}\,\ell\,n\}$$

Fig. 1. Memory operations and wrappers in separation logic.

new verification approach, *abductive deductive verification*, which integrates ideas from abductive inference with deductive separation logic verification to infer specifications in separation logic.

Automating separation logic. For deductive separation logic verification and automation, we follow in the footsteps of RefinedC [52]. RefinedC is a recently developed separation logic verification technique for establishing functional correctness of C code. Its distinguishing feature is that it is foundational and, additionally, automated: Embedded into the Coq proof assistant, RefinedC (1) provides powerful automation of separation logic, (2) inherits support for a large variety of functional correctness reasoning from Coq's ambient meta-logic, and (3) is proven sound against Caesium, a detailed model of the C semantics in Coq. For Quiver, we take inspiration from RefinedC's approach to separation logic proof automation (*i.e.*, goal-directed proof search for weakest preconditions; see §2), its separation logic-based type system for handling the complexities of C, and its embedding into Coq to support a large variety of mathematical theories.

As mentioned above, a weak spot of RefinedC is that it—like other deductive verification techniques—requires considerable amounts of specification. To illustrate this point, let us consider a poster child example for specification inference:

```
1  [[rc::parameters(n: Z)]]
2  [[rc::args(n @ int<size_t>)]]
3  [[rc::exists(l : loc)]]
4  [[rc::returns(l @ &own<uninit<n>>)]]
5  [[rc::ensures(block l n)]]
6  void *xmalloc(size_t size) {
7    void *ptr = malloc(size);
8    if (ptr == NULL) abort();
9    return ptr; }
```

```
10 [[rc::parameters(n: Z)]]
11 [[rc::args(n @ int<size_t>)]]
12 [[rc::exists(l : loc)]]
13 [[rc::returns(l @ &own<zeros<n>>)]]
14 [[rc::ensures(block l n)]]
15 void *xzalloc(size_t size) {
16   void *ptr = xmalloc(size);
17   memset(ptr, 0, size);
18   return ptr; }
```

The functions `xmalloc` and `xzalloc` are simple helper functions for wrapping memory allocation in C (inspired by similar wrappers in popular open source projects [15, 22, 48]). They encapsulate common patterns such as (1) handling the case that allocation fails and `malloc` returns `NULL` (`xmalloc`) and (2) initializing freshly allocated memory with zeros (`xzalloc`). The implementations of the two functions are dead simple. Yet, when verifying them in RefinedC, we end up writing more lines of specification (Lines 1-5 and Lines 10-14) than there are lines of code. And for no good reason: as we will see below, the specifications of `xmalloc` and `xzalloc` can be inferred from those of `malloc`, `memset`, and `abort`.

Abductive inference. A key building block for us in reducing the specification burden is the idea of *abductive inference* in the sense that we infer the specification for a piece of code by "puzzling together" existing specifications for its component parts. To illustrate this idea, let us assemble the specifications of `xmalloc` and `xzalloc` from the auxiliary operations `malloc`, `memset`, and `abort`. The specifications of all operations are depicted in Fig. 1. (For simplicity, we phrase these specifications

in a separation logic instead of RefinedC's type system.) The operation `malloc` takes a `size_t` integer n and returns either `NULL` or a pointer ℓ to a memory block of size n (denoted block $\ell\, n$) whose contents w are uninitialized (denoted $\mathsf{uninit}(w, n)$). The operation `memset`, called with zero, initializes the contents of a pointer with zeros (denoted $\mathsf{zeros}(w, n)$), and the operation `abort` never returns (postcondition False). Using these specifications, we can assemble the specification of `xmalloc` (and analogously `xzalloc`) as follows: The precondition $n \in$ `size_t` is inherited from `malloc`. The postcondition is derived from the post of `malloc` knowing that, in the `NULL`-case, we never return due to `abort`.

The idea of using abductive inference in separation logic is not new. It was first pioneered by *bi-abduction* [5, 6], a landmark technique for compositional shape analysis based on separation logic. Bi-abduction is one of the cornerstones of Meta's Infer tool for detecting bugs in millions of lines of code [4] and also inspired a line of research on bug finding using incorrectness logic [40, 47, 32]. It takes as input the code of a function and generates a separation logic specification that summarizes the footprint of the code via abductive inference. However, in the interest of supporting "push-button" automation, bi-abduction focuses on fixed, restricted fragments of separation logic. For example, the original work of Calcagno et al. [5, 6] restricts attention to points-to assertions $\ell \mapsto v$, list segments $\mathsf{lseg}(\ell, r)$, and equalities $v = v'$. As such, it cannot express—or abductively infer—for example, the specifications of `xmalloc` and `xzalloc` in Fig. 1, since they go beyond this fragment.[1]

Abductive deductive verification. With Quiver, we pursue a fundamentally different approach. Rather than trying to build push-button automation by restricting the separation logic fragment, we instead aim to integrate abductive inference into deductive verification approaches that already handle rich fragments of separation logic. To do so, we introduce a new technique we call *abductive deductive verification*. The main abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ marries deductive separation logic verification, via the *weakest precondition* connective $\mathbf{wp}\ e\ \{\Phi\}$, with abductive inference of a precondition R, via the *abduction judgment* $\Delta * [R] \vdash G$. Concretely, deriving $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ corresponds to deductively verifying the expression e in the context Δ while, simultaneously, abductively inferring any missing resources R that are needed to do so. By combining both styles of reasoning, we maintain the ability to deductively verify programs with rich separation logic specifications while additionally benefiting from the advantages of specification inference (*e.g.*, inferring the specification of `xmalloc` while verifying it; see §6.2).

Specification sketches. Since Quiver targets rich separation logics, *fully automatically* puzzling together specifications is not always the right choice (or even feasible). Consider the following extension of the previous example—a function that allocates a vector initialized with zeros:

```
20 vec_t mkvec(int n) {
21   size_t s = sizeof(int) * (size_t)n; vec_t vec = xmalloc(sizeof(struct vector));
22   vec->data = xzalloc(s); vec->len = n; [[q::type(? @ vec_t)]] return vec; }
```

(For now, we ask the reader to ignore the annotation "`[[q::...]]`".) A standard functional correctness specification for `mkvec` would be $\{n \in \texttt{int} * n \geq 0\}\ \texttt{mkvec}(n)\ \{v.\ \mathsf{vec}(v, 0^n)\}$ where $\mathsf{vec}(v, xs)$ is an abstract predicate for vectors with contents xs (a list of integers) and 0^n is a list filled with n zeros. If we simply "puzzle together" a specification for `mkvec` based on the specifications of `xmalloc` and `xzalloc` (in Fig. 1), we would arrive at a low-level specification in terms of points-to assertions and the zeros-predicate—not a high-level specification about vectors. The underlying issue is that a single function can have multiple specifications at different levels of abstraction—depending on the intent of the developer. To guide the inference to the desired one, we thus use *specification sketches*.

[1] Modifying this fragment is a challenging feat. Considerable follow-on work has gone into adding *individual extensions* (*e.g.*, linear integer arithmetic [57] or low-level pointer representation [25]). See §7 for an overview.

Guided specification inference à la Quiver. That is, Quiver explores the middle ground in between (a) taking a complete specification as user input and verifying the code (as in RefinedC) and (b) taking only the code as input and inferring the entire specification (as in bi-abduction). We take a *specification sketch* as input, use it to resolve ambiguity, and complete it to a full specification—but without requiring the user to provide every little detail. Quiver works in three steps:

(1) *Data type declarations.* First, the user defines their custom data types that are used in the code (*e.g.*, arrays, linked lists, maps, buffers, vectors, *etc.*). This step includes choosing mathematical domains, imposing invariants on values, and relating mathematical and physical representations.
(2) *Function specification sketches.* Second, the user can provide sketches for functions. These sketches are similar to separation logic specifications (*e.g.*, describing the abstract predicates for arguments). There is, however, a crucial difference: they are *incomplete* with holes for, *e.g.*, arguments of abstract predicates, additional constraints, and missing ownership.
(3) *Specification inference.* Finally, Quiver takes this sketch and completes it into a specification for the entire function using abductive deductive verification. This includes adding missing preconditions, making imprecise annotations precise, adding constraints for unspecified function arguments, and figuring out the postcondition of the function.

In the resulting system, users control how much specification they want to provide. By default, if the inference is successful, the resulting specification closely follows the code. If the user decides to "sprinkle in" some annotations that constrain function arguments or local variables to a certain data type, Quiver takes these into account and adjusts the specification accordingly. And if the user provides the complete function specification, Quiver turns into a traditional technique for verifying functional correctness. For example, the specifications of `xmalloc` and `xzalloc` can be derived fully automatically without any sketches. For `mkvec`, we only add the sketch in Line 22: it instructs Quiver that the return value is a vector, which results in the high-level vec-specification.

Contributions. Our key theoretical contribution is **Abductive Deductive Verification** (§2), which provides a powerful basis for specification inference in rich separation logics. With the abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$, we marry traditional deductive verification via the *weakest precondition* $\mathbf{wp}\ e\ \{\Phi\}$ with abductive inference via the *abduction judgment* $\Delta * [R] \vdash G$. Our key technical contribution is that we realize abductive deductive verification in the form of **Quiver**, which consists of four parts:

- The *abduction engine* **Argon** (§3), which automates the abduction judgment $\Delta * [R] \vdash G$. Its key contribution is a *goal-directed proof search procedure* for abductive inference. It supports predicate-transformer style reasoning, necessitated by the weakest precondition $\mathbf{wp}\ e\ \{\Phi\}$, provides extensible proof search, and has powerful support for instantiating existential quantifiers.
- The *type system* **Thorium** (§4), which uses types in separation logic (*à la* RefinedC) to scale automated reasoning about the weakest precondition $\mathbf{wp}\ e\ \{\Phi\}$ to the complexities of C. Its key contribution is that it works under *incomplete information* about the proof context Δ, meaning it works even when the types that are supposed to guide the proof search are yet to be determined.
- A proof-of-concept **Implementation** (§5) in the Coq proof assistant [13]—with a frontend for C, building on Iris [29] and RefinedC's Caesium semantics for C. The implementation infers specifications and, at the same time, proves them correct in Coq, which includes proving the absence of out-of-bounds accesses, use-after-free, and integer overflows.
- An **Evaluation** (§6), applying Quiver to several interesting case studies, including a dynamically-allocated vector data type and code from popular open-source libraries.

We provide the implementation and all inferred specifications in the accompanying Coq development, along with an appendix containing further details on the inferred specifications [56].

$$\begin{array}{lrll} \text{Expressions} & e & ::= & x \mid v \mid e_1 \oplus e_2 \mid f(\vec{e}) \mid e.m \mid e_1.m \leftarrow e_2 \mid \mathsf{new}() \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \mid \cdots \\ \text{Assertions} & P, Q & ::= & \ell \mapsto t \mid \mathrm{loc}(v, \ell) \mid \mathrm{int}(v, n) \mid \mathrm{P}(v, \vec{x}) \mid n_1 \le n_2 \mid n_1 = n_2 \mid \cdots \end{array}$$

Fig. 2. The exposition language λ_{expo}.

```
init(r, a, b)      := r.s ← a; r.e ← b; assert(range(r, ?, ?))
mk_range(a, b)     := let r = new() in init(r, a, b); r
size(r)            := assert(range(r, ?, ?)); r.e − r.s; assert(range(r, ?, ?))
```

Fig. 3. The implementation of the range data type. Quiver assertion annotations in blue.

Limitations. Quiver does not infer loop invariants, but supports manually provided invariants. Quiver does not infer specifications of function pointers and only handles sequential code. Moreover, while Quiver builds on the detailed Caesium C semantics, Quiver does not handle all features of C. In particular, it neither enforces that pointer accesses are aligned, nor supports unions, nor supports integer-pointer casts, and it inherits the limitations of Caesium (*e.g.*, no floating point numbers).

2 KEY IDEA: ABDUCTIVE DEDUCTIVE VERIFICATION

In this section, we explain our approach of *abductive deductive verification*. Concretely, we discuss the abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ (in §2.1), the treatment of existential quantification (in §2.2), and how we steer the inference via specification sketches (in §2.3). To avoid getting bogged down in the details of C, we focus on a simple, expository language λ_{expo} for this explanation. From §3 onwards, we will then explain how we scale the approach to actual C code.

A running example. The language λ_{expo} is depicted in Fig. 2. It is a simple, substitution-based language with heap-allocated data structures, modeled as mutable, finite maps t from fields to values (similar to objects in JavaScript). We write $\mathrm{dom}\, t$ for the fields of t and ϵ for the empty map. Values v can be locations ℓ and integers n. The expression $\mathsf{new}()$ allocates an empty struct, $e.m \leftarrow e'$ assigns e' to the field m of e, and $e.m$ dereferences field m of e. We abbreviate $e_1; e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

To reason about λ_{expo} in separation logic, we use *resources* and *pure assertions*. The resources are *points-to assertions* $\ell \mapsto t$, which assert *ownership* of a struct at location ℓ with at least the fields t, and *abstract predicates* $\mathrm{P}(v, \vec{x})$ (see §2.3). The pure assertions include $\mathrm{loc}(v, \ell)$ for "v is the location ℓ" and $\mathrm{int}(v, n)$ for "v is the integer n"; they state facts without ownership and can thus be duplicated.

To keep matters concrete, we focus on an example in λ_{expo}, *a data type for integer ranges* $[s, e)$. This range data type is represented by a struct with two integer fields: s (for the start of the range) and e (for one past the end of the range). We define three operations operating on ranges, depicted in Fig. 3: init for initializing a previously allocated range r with bounds a and b, mk_range for allocating and initializing a new range from a to b, and size to determine the size of a range r.

2.1 The Essence of Abductive Deductive Verification

We start by explaining the abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$. It consists of two parts: a separation logic *weakest precondition* $\mathbf{wp}\ e\ \{\Phi\}$ [17, 29, 28] and an *abduction judgment* $\Delta * [R] \vdash G$ where Δ is a separation logic context, R is an additional *inferred* precondition, and G is the current goal. The basic idea is that when we derive $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$, we prove that Δ together with R is a sufficient precondition for e to satisfy the postcondition Φ. That is:

$$\Delta * [R] \vdash \mathbf{wp}\ e\ \{v.\, \Phi\, v\} \quad \textit{implies} \quad \{\Delta * R\}\ e\ \{v.\, \Phi\, v\}$$

Weakest Precondition Rules (Language-Specific)

$$\mathbf{wp}\ v\ \{\Phi\} \dashv\ \Phi\, v \qquad \text{(WP-VAL)}$$

$$\mathbf{wp}\ (\text{let}\, x = e_1\, \text{in}\, e_2)\ \{\Phi\} \dashv\ \mathbf{wp}\ e_1\ \{v.\ \mathbf{wp}\ (e_2[v/x])\ \{\Phi\}\} \qquad \text{(WP-LET)}$$

$$\mathbf{wp}\ (v.m \leftarrow w)\ \{\Phi\} \dashv\ \ell \mapsto t * (\ell \mapsto t[m := w] \mathrel{-\!\!*} \Phi\,\langle\rangle)\ \textit{when}\ \text{loc}(v,\ell) * \ell \mapsto t \qquad \text{(WP-ASSIGN)}$$

$$\mathbf{wp}\ (v.m \leftarrow w)\ \{\Phi\} \dashv\ \exists \ell, t.\, \text{loc}(v,\ell) * \ell \mapsto t * (\ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\ (v.m \leftarrow w)\ \{\Phi\}) \qquad \text{(WP-ASSIGN-DEF)}$$

$$\mathbf{wp}\ (v.m)\ \{\Phi\} \dashv\ \ell \mapsto t * (\ell \mapsto t \mathrel{-\!\!*} \Phi\, t.m)\ \textit{when}\ \text{loc}(v,\ell) * \ell \mapsto t * m \in \text{dom}\, t \qquad \text{(WP-READ)}$$

$$\mathbf{wp}\ \text{new}()\ \{\Phi\} \dashv\ \forall \ell, v.\, \text{loc}(v,\ell) * \ell \mapsto \epsilon \mathrel{-\!\!*} \Phi\, v \qquad \text{(WP-NEW)}$$

$$\mathbf{wp}\ f(\vec{v})\ \{\Phi\} \dashv\ \mathbf{apply}(T\,\vec{v})\{\Phi\}\ \textit{when}\ \text{spec}(f,T) \qquad \text{(WP-CALL)}$$

$$\mathbf{wp}\ (\text{assert}(\exists x.\, P\, x))\ \{\Phi\} \dashv\ \mathbf{assert}(x.\, P\, x)\{\Phi\,\langle\rangle\} \qquad \text{(WP-ASSERT)}$$

Abduction Rules (Generic)

ABD-EMBED
$$\frac{E \dashv G\ \textit{when}\ P \qquad \Delta \vdash P \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash E}$$

ABD-RES-CTX
$$\frac{\Delta = \Delta', M \qquad \Delta' * [R] \vdash G}{\Delta * [R] \vdash M * G}$$

ABD-RES-MISSING
$$\frac{\Delta * [R] \vdash G}{\Delta * [M * R] \vdash M * G}$$

ABD-PURE-PROVE
$$\frac{\Delta \vdash \phi \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash \phi * G}$$

ABD-PURE-MISSING
$$\frac{\Delta, \phi * [R] \vdash G}{\Delta * [\phi * R] \vdash \phi * G}$$

ABD-WAND-RES
$$\frac{\Delta, M * [R] \vdash G}{\Delta * [R] \vdash M \mathrel{-\!\!*} G}$$

ABD-WAND-PURE
$$\frac{\Delta, \phi * [R] \vdash G}{\Delta * [R] \vdash \phi \mathrel{-\!\!*} G}$$

ABD-EXISTS
$$\frac{\forall x.\ (\Delta * [Rx] \vdash Gx)}{\Delta * [\exists x.\, Rx] \vdash \exists x.\, Gx}$$

ABD-ALL
$$\frac{\forall x.\ (\Delta * [R] \vdash Gx)}{\Delta * [R] \vdash \forall x.\, Gx}$$

ABD-END
$$\Delta * [\forall \vec{x}.\, \Delta \mathrel{-\!\!*} \Phi v] \vdash \Phi v$$

ABD-TRUE
$$\Delta * [\mathbf{True}] \vdash \mathbf{True}$$

Fig. 4. Weakest precondition rules for λ_{expo} and generic abduction rules. Overlapping weakest precondition rules are applied top-to-bottom, and overlapping abduction rules are applied left-to-right.

We explain how $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ works with the rules in Fig. 4. To stage the explanation, we present it in three steps, moving from verification to inference. In Version 1, we use the judgment for ordinary verification—without any inference—and explain the proof search strategy underlying our automation: goal-directed proof search for weakest preconditions. In Version 2, we extend the judgment to infer preconditions, by incorporating abduction into our goal-directed proof search. In Version 3, we extend it further to infer complete specifications. In this last version, we explain why we infer so-called "predicate transformer specifications" instead of Hoare triples.

To keep matters concrete, we focus on the operation init. For now, we ask the reader to ignore the blue assertion in the code of init (in Fig. 3). We will infer the following specification for init:

$$\{\text{loc}(v_r,\ell) * \ell \mapsto t\}\ \text{init}(v_r, v_a, v_b)\ \{\_.\ \text{loc}(v_r,\ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]\}$$

The precondition assumes that v_r is a location ℓ and that this location ℓ stores a struct with contents t. The postcondition ensures that ℓ stores an updated struct with v_a in its s-field and v_b in its e-field.

Version 1: Deductive verification. Say we are given the spec $\{P_{\texttt{init}}\}\ \text{init}(v_r, v_a, v_b)\ \{\_.\ Q_{\texttt{init}}\}$ with $P_{\texttt{init}} \triangleq \text{loc}(v_r,\ell) * \ell \mapsto t$ and $Q_{\texttt{init}} \triangleq \text{loc}(v_r,\ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]$. We aim to deductively verify it via the judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$, where we instantiate $\Delta \triangleq P_{\texttt{init}}$, $\Phi \triangleq Q_{\texttt{init}}$, and e with the body of $\text{init}(v_r, v_a, v_b)$, and for now we ignore the precondition R (pretend it is **True**).

Our proof strategy for deductive verification—following in the footsteps of RefinedC [52]—is to employ *goal-directed proof search for weakest preconditions*. It is goal-directed in the sense that, to derive $\Delta * [\_] \vdash G$, we iteratively inspect the current goal G and then apply a matching rule that

transforms G into a new goal G'. (If multiple rules match, we use the first one whose side conditions can be proven; the order is described in the figures.) And to reason about weakest preconditions, we use the rule ABD-EMBED to embed deductive proof rules of the form "$E \dashv G$ *when* P": here, E is the weakest-pre goal we are trying to solve, and G is the new subgoal that implies it under the (optional) side condition P. In the case of $\mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a};\ v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ Q_{\mathtt{init}}\}$, we start by using ABD-EMBED to apply WP-LET. It breaks up sequential composition $e_1; e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$ by putting the **wp** of e_2 into the post of e_1, turning the goal into $\mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a})\ \{\_.\ \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ Q_{\mathtt{init}}\}\}$.

Next, we apply WP-ASSIGN. It imposes an additional side condition on the context (via ABD-EMBED), namely v_r should be some location ℓ for which we have a points-to assertion $\ell \mapsto t$. Thus, leaving:

$$\mathrm{loc}(v_\mathsf{r}, \ell),\ \ell \mapsto t * [\_] \vdash \ell \mapsto t * (\ell \mapsto t[\mathsf{s} := v_\mathsf{a}] \mathrel{-\!\!*} \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ Q_{\mathtt{init}}\})$$

The rule WP-ASSIGN has transformed the goal such that we should first give up the ownership of ℓ (with "$\ell \mapsto t *$ ") and then we get back the updated ownership again (with "$\ell \mapsto t[\mathsf{s} := v_\mathsf{a}] \mathrel{-\!\!*}$ "). We do the former with ABD-RES-CTX and the latter with ABD-WAND-RES, leaving us to prove

$$\mathrm{loc}(v_\mathsf{r}, \ell),\ \ell \mapsto t[\mathsf{s} := v_\mathsf{a}] * [\_] \vdash \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ Q_{\mathtt{init}}\}$$

We proceed in a similar fashion for the second assignment, updating the points-to assertion to $\ell \mapsto t[\mathsf{s} := v_\mathsf{a},\ \mathsf{e} := v_\mathsf{b}]$, which satisfies the desired postcondition.

Version 2: Abducting the precondition. We now turn to *inference*. As before, we deductively verify $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ in goal-directed fashion—except that now we allow for the possibility that the context Δ was not sufficient to prove the goal, and we infer the missing precondition R. This means that, whereas Δ, e, and Φ are inputs, R is an *output*. To stage the presentation, we will make the simplifying assumption in this version that the post Φ is simply **True**.

To infer the precondition $P_{\mathtt{init}}$, we solve $\emptyset * [R] \vdash \mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a};\ v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ \mathbf{True}\}$ where the context is empty. The proof search proceeds as in Version 1 until we reach the first assignment:

$$\emptyset * [R] \vdash \mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a})\ \{\_.\ \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ \mathbf{True}\}\}$$

At this point, the rule WP-ASSIGN does not apply anymore, since the ownership of v_r is not in the context. (It is empty!) Instead, the missing ownership should come from the precondition that we want to infer, so we must *add it* to the precondition R. To do so, we proceed in several steps. First, we use a second rule for assignment, WP-ASSIGN-DEF, resulting in:

$$\emptyset * [R] \vdash \exists \ell, t.\ \mathrm{loc}(v_\mathsf{r}, \ell) * \ell \mapsto t * (\ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a})\ \{\_.\ \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ \mathbf{True}\}\})$$

Next, since the goal has become an existential quantifier, we apply the rule for existential quantification, ABD-EXISTS (twice). It adds existential quantifiers for ℓ and t to the precondition R. Subsequently, since $\mathrm{loc}(v_\mathsf{r}, \ell)$ and $\ell \mapsto t$ are not contained in the context $\emptyset$, they are also added to the precondition R—with ABD-PURE-MISSING for $\mathrm{loc}(v_\mathsf{r}, \ell)$ and ABD-RES-MISSING for $\ell \mapsto t$. We are left to derive

$$\mathrm{loc}(v_\mathsf{r}, \ell) * [R'] \vdash \ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\ (v_\mathsf{r}.\mathsf{s} \leftarrow v_\mathsf{a})\ \{\_.\ \mathbf{wp}\ (v_\mathsf{r}.\mathsf{e} \leftarrow v_\mathsf{b})\ \{\_.\ \mathbf{True}\}\}$$

and have already constructed $R \triangleq \exists \ell, t.\ \mathrm{loc}(v_\mathsf{r}, \ell) * \ell \mapsto t * R'$ for some R' that is yet to be determined. From here, the derivation proceeds as in Version 1 until we eventually arrive at the post **True**, facing $\mathrm{loc}(v_\mathsf{r}, \ell),\ \ell \mapsto t[\mathsf{s} := v_\mathsf{a},\ \mathsf{e} := v_\mathsf{b}] * [R'] \vdash \mathbf{True}$. We finish with ABD-TRUE, resolving $R' \triangleq \mathbf{True}$. Thus, we have inferred the pre $R = (\exists \ell, t.\ \mathrm{loc}(v_\mathsf{r}, \ell) * \ell \mapsto t)$ such that $\{R\}\ \mathsf{init}(v_\mathsf{r}, v_\mathsf{a}, v_\mathsf{b})\ \{\_.\ \mathbf{True}\}$.

Version 3: Abducting the postcondition. Next, let us infer the *postcondition*. Intuitively, the postcondition should be the context at the end of the derivation (*i.e.*, "Δ" in ABD-TRUE). The judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$, however, does not have an *output* for the postcondition, only for the precondition R. The reason is that such a dedicated postcondition output is not needed—rather, we can encode the post as part of the pre by expressing specifications as *predicate transformers*.

A predicate transformer is a function T from postconditions Φ to preconditions $T\,\Phi$ such that $\forall\Phi.\ \{T\,\Phi\}\ e\ \{\Phi\}$. Predicate transformers are an alternative to Hoare triples for specifying functions, where we use existential quantification (+ separating conjunctions) to express preconditions, and universal quantification (+ magic wands) to express postconditions. We use colors to highlight the precondition parts (in light blue) and postcondition parts (in violet). For example, $\{\mathrm{loc}(v_r, \ell) * \ell \mapsto t\}\ \mathsf{init}(v_r, v_a, v_b)\ \{\_.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]\}$ can be expressed as:

$$T_{\mathsf{init}}(\Phi) \triangleq \underbrace{\exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t}_{\text{precondition}} * (\underbrace{\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]}_{\text{postcondition}} \mathrel{-\!\!*} \Phi\,\langle\rangle)$$

In this case, the value v_r should be a location ℓ storing a struct with contents t (precondition), and afterwards the location ℓ stores the updated contents $t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]$ (postcondition).

To *infer* a predicate transformer specification, we treat the predicate Φ abstractly during the abduction. Then, the resulting precondition $R(\Phi)$ is a predicate transformer, because

$$\forall\Phi.\ \emptyset * [R\,\Phi] \vdash \mathbf{wp}\ e\ \{v.\ \Phi\,v\} \quad \textit{implies} \quad \forall\Phi.\ \{R\,\Phi\}\ e\ \{v.\ \Phi\,v\}$$

In the case of $\mathsf{init}(v_r, v_a, v_b)$, we abduct $\emptyset * [R] \vdash \mathbf{wp}\ \mathsf{init}(v_r, v_a, v_b)\ \{v.\ \Phi\,v\}$ where Φ is abstract. We eventually hit the postcondition $\Phi\,\langle\rangle$ (in place of "**True**" in Version 2). At this point, we face $\mathrm{loc}(v_r, \ell),\ \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] * [R'] \vdash \Phi\,\langle\rangle$. To finish the derivation, we "revert" the context before the post with ABD-END ("$\forall\vec{x}$" explained below), resulting in the solution $R'(\Phi) \triangleq \mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] \mathrel{-\!\!*} \Phi\,\langle\rangle$. Plugging this into the top-level inferred specification $R(\Phi) \triangleq \exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * R'(\Phi)$, we see that $R(\Phi)$ coincides exactly with the specification $T_{\mathsf{init}}(\Phi)$. Thus, we have inferred the specification $\{P_{\mathsf{init}}\}\ \mathsf{init}(v_r, v_a, v_b)\ \{\_.\ Q_{\mathsf{init}}\}$—stated as a predicate transformer instead of a Hoare triple.

2.2 Existential Quantification

The goal-directed proof search presented above is overly simplistic in a key dimension: its treatment of existential quantification. For existential quantification, there are really two options: (1) lift the quantifier to the precondition (as above) or (2) instantiate the quantifier. For abductive deductive verification to work, we need both options. With init, we have already seen a case where we *must* lift the existential quantifier to the precondition, because the context $\emptyset$ is empty (in §2.1). To illustrate when we want the second option, we consider the function mk_range.

For mk_range, we want to infer the following specification:

$$T_{\mathsf{mk\_range}}(\Phi) \triangleq \mathbf{True} * (\forall \ell, v_r.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto \{\ \mathsf{s} = v_a; \mathsf{e} = v_b\ \} \mathrel{-\!\!*} \Phi\,v_r)$$

The precondition is **True** and the postcondition ensures that the return value v_r is a location ℓ storing a correctly initialized range. To understand why inferring this specification requires existential instantiation, we unfold the weakest precondition semantics of $\mathbf{wp}\ \mathsf{mk\_range}(v_a, v_b)\ \{\Phi\}$ because it reveals the existential quantifier that we must instantiate. We write $P \vDash Q$ for semantic entailment. Specifically, we unfold mk_range, the let-binding, the allocation new(), and the init-call:

$$\begin{aligned}
&\mathbf{wp}\ \mathsf{mk_range}(v_a, v_b)\ \{v.\ \Phi\,v\} \\
\dashv\ &\mathbf{wp}\ \mathsf{let\ r = new()\ in\ init(r}, v_a, v_b); \mathsf{\ r}\ \{v.\ \Phi\,v\} \\
\dashv\ &\mathbf{wp}\ \mathsf{new()}\ \{v_r.\ \mathbf{wp}\ \mathsf{init}(v_r, v_a, v_b)\ \{_.\ \Phi\,v_r\}\} \\
\dashv\ &\underbrace{\forall \ell, v_r.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto \epsilon}_{\text{from new()}} \mathrel{-\!\!*} \underbrace{\exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * (\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] \mathrel{-\!\!*} \Phi\,v_r)}_{\text{from init}(v_r, v_a, v_b)}
\end{aligned}$$

Note the existential "$\exists \ell, t.$ " arising from the call to init. We should not lift this quantifier into the precondition, because its value *depends* on the result of allocating a fresh location inside mk_range. Instead, we should *instantiate* this quantifier with the location ℓ obtained from new() (and t with ϵ).

$$\mathbf{ex}(x.\, \exists y.\, G\, x\, y) \;\dashv\; \mathbf{ex}(x, y.\, G\, x\, y) \qquad \textsc{(ex-ex)}$$
$$\mathbf{ex}(x, y.\, y = o * G\, x\, y) \;\dashv\; \mathbf{ex}(x.\, G\, o\, y) \qquad \textsc{(ex-inst)}$$
$$\mathbf{ex}(x, y.\, \mathrm{loc}(v, y) * G\, x\, y) \;\dashv\; \mathbf{ex}(x, y.\, y = \ell * G\, x\, y) \;\; \textit{when}\; \mathrm{loc}(v, \ell) \qquad \textsc{(ex-loc)}$$
$$\mathbf{ex}(x, y.\, \ell \mapsto x * G\, x\, y) \;\dashv\; \ell \mapsto t * \mathbf{ex}(x, y.\, x = t * G\, x\, y) \;\; \textit{when}\; \ell \mapsto t \qquad \textsc{(ex-pointsto)}$$
$$\mathbf{ex}(x.\, \mathrm{P}(v, \vec{y}) * G\, x) \;\dashv\; \mathbf{ex}(x.\, Q * G\, x) \;\; \textit{when}\; \mathsf{known}(v) * \mathrm{P}(v, \vec{y}) \triangleq Q \qquad \textsc{(ex-unfold)}$$
$$\mathbf{ex}(x.\, \mathrm{P}(v, \vec{y}) * G\, x) \;\dashv\; \exists \vec{z}.\, \mathrm{P}(v, \vec{z}) * \mathbf{ex}(x.\, \vec{y} = \vec{z} * G\, x) \qquad \textsc{(ex-pred)}$$

Fig. 5. Existential instantiation rules for λ_{expo} (simplified), where $\mathsf{known}(v) \triangleq (\exists \ell.\, \mathrm{loc}(v, \ell)) \vee (\exists n.\, \mathrm{int}(v, n))$.

Instantiating existentials. To instantiate existential quantifiers during goal-directed proof search, we introduce a new goal $\mathbf{ex}(x.\, G\, x)$ (semantically $\exists x.\, G\, x \vDash \mathbf{ex}(x.\, G\, x)$). It is triggered for function calls $f(\vec{v})$ to instantiate existential quantifiers in the spec of f. That is, for function calls $f(\vec{v})$, we use WP-CALL, which looks for a specification T for f (by searching for $\mathrm{spec}(f, T)$) and then creates the function application goal "$\mathbf{apply}(T)\{\Phi\}$" (semantically $T\, \Phi \vDash \mathbf{apply}(T)\{\Phi\}$). The proof search treats $\mathbf{apply}(T)\{\Phi\}$ roughly as "$\mathbf{ex}(\_.\, T\, \Phi)$" (see §3) to instantiate existentials in T.

For example, in the inference of mk_range, we eventually encounter the goal:

$$\mathrm{loc}(v_{\mathsf{r}}, \ell),\; \ell \mapsto \epsilon * [R] \vdash \mathbf{ex}(\_.\, \exists x, y.\, \mathrm{loc}(v_{\mathsf{r}}, x) * x \mapsto y * G_{\mathtt{rest}}\, x\, y)$$

where **ex** contains the precondition part of $T_{\mathtt{init}}$ and we summarize the remainder as "$G_{\mathtt{rest}}$". A simplified selection of rules for $\mathbf{ex}(x.\, G\, x)$ is depicted in Fig. 5 (see §3). In this case, we proceed by (1) gathering existentials (with EX-EX), $\mathbf{ex}(x, y.\, \mathrm{loc}(v_{\mathsf{r}}, x) * x \mapsto y * G_{\mathtt{rest}}\, x\, y)$; (2) matching the location ℓ based on the context (with EX-LOC), $\mathbf{ex}(x, y.\, x = \ell * x \mapsto y * G_{\mathtt{rest}}\, x\, y)$; (3) instantiating x based on equality (with EX-INST), $\mathbf{ex}(y.\, \ell \mapsto y * G_{\mathtt{rest}}\, \ell\, y)$; (4) matching the points-to assertion (with EX-POINTSTO), $\ell \mapsto \epsilon * \mathbf{ex}(y.\, y = \epsilon * G_{\mathtt{rest}}\, \ell\, y)$; and (5) instantiating y based on equality (with EX-INST), $\mathbf{ex}(\_.\, G_{\mathtt{rest}}\, \ell\, \epsilon)$. In doing so, we arrive at the desired specification $T_{\mathtt{mk\_range}}(\Phi)$.

∃∀-specifications. One may wonder why we bother with existential quantifier instantiation and do not use the predicate transformer from semantically unfolding $\mathbf{wp}\ \mathsf{mk\_range}(v_{\mathsf{a}}, v_{\mathsf{b}})\ \{v.\, \Phi\, v\}$,

$$\forall \ell, v_{\mathsf{r}}.\, \mathrm{loc}(v_{\mathsf{r}}, \ell) * \ell \mapsto \epsilon \mathrel{-\!\!*} \exists \ell, t.\, \mathrm{loc}(v_{\mathsf{r}}, \ell) * \ell \mapsto t * (\mathrm{loc}(v_{\mathsf{r}}, \ell) * \ell \mapsto t\,[\mathsf{s} := v_{\mathsf{a}},\, \mathsf{e} := v_{\mathsf{b}}] \mathrel{-\!\!*} \Phi\, v_{\mathsf{r}})$$

as the specification for mk_range. The underlying issue is that while predicate transformers can, in principle, contain arbitrary quantifier alternations, a predicate transformer T that goes beyond a single ∃∀-alternation can barely be considered a specification: it alternates preconditions (∃) with postconditions (∀), thus making it difficult to understand what T *means* as a specification. Therefore, a key design decision of Quiver is to restrict ourselves to a single ∃∀-alternation.

In fact, this is *enforced* by an asymmetry in our quantifier rules: ABD-EXISTS adds existential quantifiers to the precondition, but ABD-ALL does not add universal quantifiers; it only introduces them in the goal. The rule that adds universal quantifiers to the precondition is ABD-END, used only at the very end. It adds those universal quantifiers that have been introduced by ABD-ALL in the goal and are potentially now contained in the context Δ. The key benefit of a single ∃∀-alternation is that, in the resulting predicate transformers, preconditions (∃) always appear before postconditions (∀).

2.3 Specification Sketches

So far, we have discussed how we can infer specifications *without any user guidance*. The resulting specifications describe "low-level" memory footprints, but they do not yet use any abstract predicates (*i.e.*, user defined predicates for data types). Abstract predicates are, however, a hallmark of separation logic verification. For instance, for the range data type, a standard approach would be to

conceal the implementation details behind a predicate $\mathrm{range}(v_r, n_s, n_e)$, which can be understood abstractly as "v_r is the range $[n_s, n_e)$". To guide the inference towards "high-level" specifications with abstract predicates, we integrate *specification sketches* into abductive deductive verification.

Specification sketches. To explain what sketches are and how we integrate them, we continue with the range example. We define the abstract predicate $\mathrm{range}(v_r, n_s, n_e)$:

$$\mathrm{range}(v_r, n_s, n_e) \triangleq \exists \ell, v_s, v_e.\, \mathrm{loc}(v, \ell) * \ell \mapsto \{\mathsf{s} = v_s; \mathsf{e} = v_e\} * \mathrm{int}(v_s, n_s) * \mathrm{int}(v_e, n_e) * 0 \le n_s \le n_e$$

The predicate ensures that the s-field and e-field are integers n_s and n_e, and that the integer bounds form a valid, non-negative range by imposing $0 \le n_s \le n_e$.

To infer specifications involving the range-predicate, we add sketches to the implementation of the range operations. A sketch is an inline assertion "assert(...)", which describes *part of the logical state* at the program point, and which may use question marks "?" to leave holes in the description. For example, in init, we add the assertion "assert(range(r, ?, ?))" to mean r is some range [?, ?) at the end of the init function. The idea is that the proof search then takes this sketch into account and adjusts the resulting specification. That is, for init, the inferred specification becomes:

$$T^{\mathsf{ran}}_{\mathtt{init}}(\Phi) \triangleq \exists \ell, t, n_a, n_b.\, \mathrm{loc}(v_r, \ell) * \ell \mapsto t * \mathrm{int}(v_a, n_a) * \mathrm{int}(v_b, n_b) * 0 \le n_a \le n_b * (\mathrm{range}(v_r, n_a, n_b) \mathrel{-\!\!*} \Phi\langle\rangle)$$

The precondition is extended by three new assumptions: $\mathrm{int}(v_a, n_a)$ and $\mathrm{int}(v_b, n_b)$ requiring v_a and v_b to be integers n_a and n_b, and $0 \le n_a \le n_b$ to impose the range constraint on the integers. The postcondition is changed to indicate that v_r stores the range $[n_a, n_b)$ after calling the function.

Abducting specification sketches. As far as abductive deductive verification is concerned, every sketch corresponds to a separation logic proposition $\exists x.\, P\, x$, with existential quantifiers for question marks. For example, "assert(range(r, ?, ?))" in init corresponds to "$\exists x, y.\, \mathrm{range}(v_r, x, y)$". We use the sketch to update the internal separation logic state at the point of the assertion. Concretely, to integrate sketches into $\Delta * [R] \vdash G$, we introduce a new goal $\mathbf{assert}(x.\, P\, x)\{G\}$. When we encounter $\mathbf{assert}(x.\, P\, x)\{G\}$, we first (1) *prove* $P\, x$ for some x—abducting additional preconditions where necessary—and, subsequently, (2) *assume* $P\, x$ for the remainder of the inference. To deal with the existential "x" in the sketch "$\exists x.\, P\, x$", we define $\mathbf{assert}(x.\, P\, x)\{G\}$ using **ex**, roughly as "$\mathbf{ex}(x.\, P\, x * (P\, x \mathrel{-\!\!*} G))$" (see §3). Here, we use the same pattern as WP-ASSIGN and WP-READ: we first produce "$P\, x$" and then assume "$P\, x$" again for the remainder of the inference.

For example, inside the derivation of init, we encounter $\mathbf{wp}\ (\mathrm{assert}(\exists x, y.\, \mathrm{range}(v_r, x, y)))\ \{\Phi\}$. We apply (WP-ASSERT) and are confronted with the new **assert**-goal:

$$\mathrm{loc}(v_r, \ell),\ \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] * [R'] \vdash \mathbf{assert}(x, y.\, \mathrm{range}(v_r, x, y))\{\Phi\,\langle\rangle\}$$

It boils down to $\mathbf{ex}(x, y.\, \mathrm{range}(v_r, x, y) * (\mathrm{range}(v_r, x, y) \mathrel{-\!\!*} \Phi\,\langle\rangle))$. The *first part*, "$\mathrm{range}(v_r, x, y)*$", is handled by (a) unfolding $\mathrm{range}(v_r, x, y)$ with EX-UNFOLD and, then, (b) abducting anything missing for proving the body of $\mathrm{range}(v_r, x, y)$—here $\mathrm{int}(v_a, n_a)\ \mathrm{int}(v_b, n_b)$, and $0 \le n_a \le n_b$—while also instantiating $x \triangleq n_a$ and $y \triangleq n_b$. The *second part*, "$\mathrm{range}(v_r, x, y) \mathrel{-\!\!*}$ ", adds $\mathrm{range}(v_r, n_a, n_b)$ to the context (ABD-WAND-RES), which then eventually ends up in the postcondition of $T^{\mathsf{ran}}_{\mathtt{init}}$ (via ABD-END).

Sketches *vs.* specifications. The other two range operations highlight two important benefits of specification sketches over full-fledged specifications. First, we can provide similar sketches for multiple functions, yet obtain different specifications. For example, for the function size, we provide the same assertion sketches as for init, yet obtain:

$$T^{\mathsf{ran}}_{\mathtt{size}}(\Phi) \triangleq \exists n_s, n_e.\, \mathrm{range}(v_r, n_s, n_e) * (\forall w.\, \mathrm{range}(v_r, n_s, n_e) * \mathrm{int}(w, n_e - n_s) \mathrel{-\!\!*} \Phi\, w)$$

The precondition $\mathrm{range}(v_r, n_s, n_e)$ arises, because unlike for init, when we encounter the first sketch in size, the context contains no information about v_r that could be used to prove $\mathrm{range}(v_r, n_s, n_e)$.

$$
\begin{array}{lrcl}
\text{State Proposition} & S & ::= & \phi \mid M \mid S_1 * S_2 \mid \exists x.\, S\, x \qquad (M \in \text{Resource}) \\
\text{Predicate Transformer} & T & ::= & \exists \vec{x}.\, S\, \vec{x} * T\, \vec{x} \mid T_1 \wedge T_2 \mid \textbf{if } \phi \textbf{ then } T_1 \textbf{ else } T_2 \mid \forall \vec{x}.\, S\, \vec{x} \mathrel{-\!\!*} \Phi\, \vec{x} \\
\text{Goal} & G & ::= & \Phi\, v \mid E \mid S * G \mid S \mathrel{-\!\!*} G \mid G_1 \wedge G_2 \mid \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2 \\
& & \mid & \exists x.\, G x \mid \forall x.\, G x \mid \textbf{ex}(x.\, G x) \mid \textbf{simpl}(T)\{G\} \mid \textbf{bind}(G_1)\{G_2\} \\
\text{Contexts} & \Delta & ::= & (\Gamma, \Omega_*, \Omega_\square) \qquad (\Gamma \in \text{List(Prop)},\ \Omega \in \text{List(Resource)})
\end{array}
$$

Fig. 6. The syntax of Argon.

$$
\textsc{abd-if-true}\ \frac{\Delta \vdash \phi \qquad \Delta * [R] \vdash G_1}{\Delta * [R] \vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2}
\qquad
\textsc{abd-if}\ \frac{\Delta, \phi * [R_1] \vdash G_1 \qquad \Delta, \neg\phi * [R_2] \vdash G_2}{\Delta * [\textbf{if } \phi \textbf{ then } R_1 \textbf{ else } R_2] \vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2}
$$

$$
\textsc{abd-conj}\ \frac{\Delta * [R_i] \vdash G_i \text{ for } i = 1, 2}{\Delta * [R_1 \wedge R_2] \vdash G_1 \wedge G_2}
\qquad
\textsc{abd-ex}\ \frac{\textbf{ex}(x.\, G_1 x \mid S\, x) \dashv G_2 \text{ when } P \qquad \Delta \vdash P \qquad \Delta * [R] \vdash G_2}{\Delta * [R] \vdash \textbf{ex}(x.\, G_1\, x \mid S\, x)}
$$

$$
\textsc{abd-simpl}\ \frac{(\forall \Phi.\, T\, \Phi \Rrightarrow T'\, \Phi) \qquad \Delta * [R] \vdash G\, T'}{\Delta * [R] \vdash \textbf{simpl}(T)\{T''.\, G\, T''\}}
\qquad
\textsc{abd-bind}\ \frac{\forall \Phi.\, \Delta * [T\, \Phi] \vdash G_1\, \Phi \qquad \Delta_\square * [R] \vdash G_2\, T}{\Delta * [R] \vdash \textbf{bind}(\Phi.\, G_1\, \Phi)\{T'.\, G_2\, T'\}}
$$

$$
\textsc{abd-fail}\ \Delta * [\textbf{False}] \vdash G
\qquad
\textsc{simplify}\ \frac{P_1 \Rrightarrow_{\text{norm}} P_2 \qquad P_2 \Rrightarrow_{\text{simp}} P_3 \qquad P_3 \Rrightarrow_{\text{ex}} P_4}{P_1 \Rrightarrow P_4}
$$

Fig. 7. Additional abduction rules, extending the abduction rules of Fig. 4.

Thus, the resource is added as a whole to the precondition (via EX-PRED and then ABD-RES-MISSING). The postcondition contains the additional information that the return value w is the integer $n_s - n_e$.

Second, abductive deductive verification is *compositional*. The sketch in init not only affects init, but also mk_range. That is, if we infer a specification of mk_range again—against the new specification $T^{\mathsf{ran}}_{\mathtt{init}}$—we obtain the following specification *without any additional sketches*:

$$T^{\mathsf{ran}}_{\mathtt{mk\_range}}(\Phi) \triangleq \exists n_a, n_b.\, \text{int}(v_a, n_a) * \text{int}(v_b, n_b) * 0 \le n_a \le n_b * (\forall v_r.\, \text{range}(v_r, n_a, n_b) \mathrel{-\!\!*} \Phi\, v_r)$$

The precondition changes to incorporate the additional assumptions on v_a and v_b, and the postcondition ensures that the return value v_r is the correctly initialized range(v_r, n_a, n_b) from init.

3 THE ABDUCTION ENGINE ARGON

Having introduced the idea of abductive deductive verification (§2), let us now focus on the first part of Quiver, the *abduction engine* **Argon**. It provides automation for the abduction judgment $\Delta * [R] \vdash G$. Intuitively, $\Delta * [R] \vdash G$ means G holds under the assumption of the context Δ and the additional precondition R. Accordingly, for a context $\Delta = (\Gamma, \Omega_*, \Omega_\square)$, we define

$$\Delta * [R] \vdash G \triangleq (\circledast_{\phi \in \Gamma} \phi) * (\circledast_{Q \in \Omega_*} Q) * (\circledast_{Q \in \Omega_\square} \square\, Q) * R \vDash G$$

The context Δ consists of three parts: pure assertions Γ (*e.g.*, $n \ge 0$ and $\text{loc}(v, \ell)$), ownership assertions Ω_*, and persistent assertions $\Omega_\square$. The assertions in Ω_* and $\Omega_\square$ are *resources* M, the basic building blocks to describe the program state (*e.g.*, $\ell \mapsto t$ and $\text{range}(v, n, m)$ in §2). The persistent resources in $\Omega_\square$ remain in the context forever while the ownership resources in Ω_* can be removed.

Goal-directed proof search. The key technique that turns Argon's inference rules into an automated abduction algorithm is *goal-directed proof search*: At each step of the abduction with state $\Delta * [R] \vdash G$, Argon matches on the goal G and applies the first rule with a matching conclusion and whose side conditions can be proven (where the order in the paper is left-to-right, top-to-bottom). After applying the rule, it then recursively proceeds with the premises. The goals that Argon supports are depicted in Fig. 6. Below, we discuss which purpose they serve, and how they are dealt with during goal-directed proof search, using the rules in Fig. 4 and Fig. 7.

Embedded goals. *Embedded goals* E sit at the heart of Argon. They embed deductive proof systems into Argon such as a weakest precondition calculus (in §2) and the type system Thorium (in §4)—using an extensible set of reasoning rules. Concretely, when Argon encounters an embedded goal E (ABD-EMBED in Fig. 4), it searches for a reasoning rule $E \dashv G$ *when* P by matching on E and continues with the goal G if the side condition P is provable in the current context Δ. Embedded goals can be weakest preconditions **wp** $e\ \{\Phi\}$, but also other auxiliary judgments (*e.g.*, Thorium introduces a judgment for "type conversion", discussed in the appendix [56]).

Separating conjunction and magic wand. We turn to *separating conjunction* $S * G$ and the *magic wand* $S \mathrel{-\!\!*} G$. The goal $S * G$ instructs Argon to prove the assertion S while $S \mathrel{-\!\!*} G$ introduces S into the context. Argon avoids ambiguity during the proof search by restricting S to *state propositions*, *i.e.*, assertions over the state of the program consisting of resources and pure assertions. (For efficient proof automation, Sammler et al. [52] employ a similar restriction, but not in the context of abduction.) We consider the two interesting cases: If $S = \phi$ is a pure assertion, we either prove ϕ through a pure entailment $\Delta \vdash \phi$ (ABD-PURE-PROVE in Fig. 4), or we add ϕ to the precondition and the context (ABD-PURE-MISSING in Fig. 4). If $S = M$ is a resource, we either find the resource M in the context (ABD-RES-CTX in Fig. 4), or we add it as a missing assertion to the precondition (ABD-RES-MISSING in Fig. 4). The other cases for S (namely, $S_1 * S_2$ and $\exists x.\, S'\, x$) are handled by straightforward rules (not shown) which serve to hoist out existential quantifiers and move a ϕ or M to the left side of the goal using associativity of separating conjunction. We adopt a similar restriction for *magic wands* $S \mathrel{-\!\!*} G$. It lets us split S into pure assertions ϕ and resources M that are then added to the context (ABD-WAND-PURE resp. ABD-WAND-RES in Fig. 4).

Conditionals and conjunctions. We turn to *conditionals* **if** ϕ **then** G_1 **else** G_2 and *conjunctions* $G_1 \wedge G_2$. For a conditional **if** ϕ **then** G_1 **else** G_2, we first try to eliminate it by proving or disproving the condition ϕ (*i.e.*, by applying ABD-IF-TRUE or the corresponding rule for $\Delta \vdash \neg\phi$). Otherwise, we lift it to the precondition (ABD-IF).[2] For *conjunctions* $G_1 \wedge G_2$, we lift the conjunction to the precondition (ABD-CONJ). Conceptually, a conjunction means the reason for the choice between both branches is *internal* to the function, *i.e.*, it cannot be influenced by the caller. For example, a call to `malloc` may return `NULL` or a valid pointer. A client cannot influence which one it is, making the predicate transformer for `malloc` a conjunction of both cases (shown in the appendix [56]).

Quantifiers and post conditions. *Existential quantification* $\exists x.\, Gx$ in the goal is resolved by adding an existential quantifier to the precondition R (ABD-EXISTS in Fig. 4) and *universal quantification* $\forall x.\, Gx$ is resolved by introducing x in the goal but leaving the precondition R unchanged (ABD-ALL in Fig. 4). Universal quantifiers are added to the precondition only when we reach the *postcondition goal* $\Phi\, v$ (ABD-END in Fig. 4). Concretely, when we encounter $\Phi\, v$, we revert those $\vec{x}$ that have been previously introduced in the goal (with ABD-ALL). They are added in front of the context Δ since the context might refer to them at this point (*e.g.*, v_r and ℓ in $T_{\texttt{mk\_range}}$ in §2.2).

[2] If the inferred preconditions R_1 and R_2 coincide, the conditional can be removed altogether by simplification. Additionally, in some cases, the inferred preconditions R_1 and R_2 can also be joined using a heuristic of the Thorium type system.

$$\mathbf{ex}(x.\ \exists y.\ G\,x\,y \mid S\,x) \dashv \mathbf{ex}(x, y.\ G\,x\,y \mid S\,x) \quad \text{(EX-EXISTS)}$$
$$\mathbf{ex}(x.\ \phi * G\,x \mid S\,x) \dashv \phi * \mathbf{ex}(x.\ G\,x \mid S\,x) \quad \text{(EX-PURE)}$$
$$\mathbf{ex}(x.\ \phi\,x * G\,x \mid S\,x) \dashv \mathbf{ex}(x.\ G\,x \mid \phi\,x * S\,x) \quad \text{(EX-PURE-BLOCKED)}$$
$$\mathbf{ex}(x, y.\ x = o * G\,x\,y \mid S\,x\,y) \dashv \mathbf{ex}(\_.\ G' \mid \mathbf{True}) \ \textit{when}\ (\exists x, y.\ x = o * S\,x\,y * G\,x\,y) \Rrightarrow G' \quad \text{(EX-EQ)}$$
$$\mathbf{ex}(x, y.\ G\,x\,y \mid S\,x\,y) \dashv \exists y.\ \mathbf{ex}(x.\ S\,x\,y * G\,x\,y \mid \mathbf{True}) \quad \text{(EX-LIFT)}$$
$$\mathbf{ex}(\_.\ G \mid \mathbf{True}) \dashv G \quad \text{(EX-DONE)}$$

Fig. 8. Selection of existential instantiation rules for $\mathbf{ex}(x.\ Gx \mid Sx)$.

Simplification. One of the main ways[3] to integrate reasoning about mathematical theories into the Argon proof search is *simplification*. It comes in two forms: (1) a general-purpose simplification judgment $P \Rrightarrow Q$ (semantically $Q \vDash P$), which simplifies P into Q and (2) a goal $\mathbf{simpl}(T)\{T'.\ G\,T'\}$ that uses the judgment $P \Rrightarrow Q$ to simplify a predicate transformer T (ABD-SIMPL). The simplification judgment $P \Rrightarrow Q$ proceeds in three steps[4] (SIMPLIFY), $P_1 \Rrightarrow_{\text{norm}} P_2 \Rrightarrow_{\text{simp}} P_3 \Rrightarrow_{\text{ex}} P_4$: First, with $P_1 \Rrightarrow_{\text{norm}} P_2$, we normalize P_1 into a normal form analogous to predicate transformers in Fig. 6 (*e.g.*, by lifting out existentials). Then, with $P_2 \Rrightarrow_{\text{simp}} P_3$, we simplify pure propositions in P_2. Finally, with $P_3 \Rrightarrow_{\text{ex}} P_4$, we instantiate existentials based on *equalities* $a = b$ in P_3. For example,

$$x \geq 0 * (\exists y.\ 4x = 4y * Ty) \Rrightarrow_{\text{norm}} \exists y.\ x \geq 0 * 4x = 4y * Ty \Rrightarrow_{\text{simp}} \exists y.\ x \geq 0 * x = y * Ty \Rrightarrow_{\text{ex}} x \geq 0 * Tx$$

Normalization lifts the existential to the outside, then simplification removes the multiplication with 4 (since $4x = 4y$ iff $x = y$), and finally instantiation resolves $y \triangleq x$ based on the equality.

As illustrated above, the simplification step $P_3 \Rrightarrow_{\text{simp}} P_4$ integrates mathematical theories. It uses (a) *pure abduction rules* $\phi \Rrightarrow_{\text{simp}} \psi$, (b) *rewriting simplification rules* $\phi[a] \Rrightarrow_{\text{simp}} \phi[b]$ if $a = b$, and (c) *solvers* $\phi \Rrightarrow_{\text{simp}} \text{True}$ if ϕ. We use simplification rules and solvers for, *e.g.*, integers, injective functions, and lists, and the simplification rules can be extended as needed.

Existential instantiation. The goal $\mathbf{ex}(x.\ G\,x)$ is used for existential instantiation. It has the following key characteristics: it is agnostic to the order of existential quantifiers in G; it is agnostic to the order of conjuncts in a separating conjunction; it inherits the simplification of $P \Rrightarrow Q$; and it allows us to destruct existential quantifiers in the context. Moreover, similar to embedded goals E, it is extensible in the sense that additional rules can be added. To achieve these characteristics, we generalize $\mathbf{ex}(x.\ G\,x)$ to the form "$\mathbf{ex}(x.\ G\,x \mid S\,x)$", where the state proposition S collects "blocked" assertions (explained below) and define $\mathbf{ex}(x.\ G\,x) \triangleq \mathbf{ex}(x.\ G\,x \mid \mathbf{True})$.

The proof search for $\mathbf{ex}(x.\ G\,x \mid S\,x)$ proceeds by applying existential instantiation rules of the form $\mathbf{ex}(x.\ G_1\,x \mid S\,x) \dashv G_2$ *when* P (ABD-EX), analogous to the rule for embedded goals (ABD-EMBED in Fig. 4). We discuss the most important rules, depicted in Fig. 8 (and omit rules such as applying associativity for separating conjunction): For *existential quantifiers* $\exists y.\ G\,y$, we add a binding (EX-EXISTS). For *pure propositions* ϕ that do not depend on x, we lift them out of the goal (EX-PURE). For pure propositions ϕ that make x precise (*e.g.*, equality), we use simplification to instantiate the existential (EX-EQ). For pure propositions that depend on x but do not lead to instantiation, we put them on the "blocked stack", meaning we add them to the state goal S (EX-PURE-BLOCKED). The blocked stack allows us to traverse further into the goal G and, thereby, we can be agnostic about

[3] Additionally, Argon allows integrating solvers for mathematical theories via $\Delta \vdash \phi$.
[4] We may apply simplification multiple times to benefit from existential instantiations for further simplification and vice versa. By default, the implementation of Quiver simplifies twice.

the order of conjuncts in a separating conjunction. A blocked assertion may become unblocked when we can instantiate an existential. Thus, EX-EQ adds the "blocked stack" back into the goal.

For *resources* M, each instantiation of Argon can handle them as desired by extending the rules of $\mathbf{ex}(x.\, G_1\, x \mid S\, x) \dashv G_2$ *when* P. For example, we have seen (simplified) rules in Fig. 5, and Thorium adds rules for *its* resources: type assignments (see §4). Finally, we have rules for when the goal G is *stuck* in the sense that no other rule applies: if there are existentials left to instantiate (EX-LIFT), we lift one of them outside and, once there are none left (EX-DONE), we continue with the goal G.

Sequential composition. The goal $\mathbf{bind}(\Phi.\, G_1\, \Phi)\{T.\, G_2\, T\}$ implements *sequential composition* of abduction goals (ABD-BIND). It works as follows: First, it will abduct G_1 for an arbitrary postcondition Φ. The result of this abduction is a predicate transformer T. Then, it will abduct G_2, passing it the newly abducted predicate transformer T as an argument. (To implement sequential composition, ABD-BIND has to avoid duplicating ownership, and therefore gives only the persistent part of the context $\Delta_\square$ to G_2.) In other words, **bind** makes abduction available *inside of an abduction*.

With **bind**, we finally have all the pieces needed to define the goals for applying predicate transformers $\mathbf{apply}(T)\{\Phi\}$ (from §2.2) and specification sketches $\mathbf{assert}(x.\, S\, x)\{\Phi\}$ (from §2.3):

$$\mathbf{apply}(T)\{\Phi\} \triangleq \mathbf{bind}(\Psi.\, \mathbf{ex}(\_.\, T\, \Psi))\{T'.\, \mathbf{simpl}(T')\{T''.\, T''\, \Phi\}\}$$
$$\mathbf{assert}(x.\, S\, x)\{\Phi\} \triangleq \mathbf{bind}(\Psi.\, \mathbf{ex}(x.\, S\, x * (S\, x \mathrel{-\!\!*} \Psi)))\{T'.\, \mathbf{simpl}(T')\{T''.\, T''\, \Phi\}\}$$

Both goals have a similar structure. For **apply**, we instantiate the existentials in T using **ex**, and for **assert**, we instantiate the existentials in the sketch "$S\, x$" using **ex**. We wrap the instantiation of existentials in the sequential composition **bind** to "cleanup" after **ex** with a simplification **simpl**. That is, since the rules for existential instantiation are extensible (ABD-EX), they can trigger arbitrary auxiliary goals, which may (indirectly) add existential quantifiers to the precondition R. By simplifying afterwards, we can potentially eliminate some of these quantifiers (*e.g.*, one goal might add "$\exists n.\ \cdots$" and another might add "$n = 0$", which is then simplified by picking $n \triangleq 0$).

Loops. Argon does not infer loop invariants, but supports loops with manually provided loop invariants (without sketches). For a given loop invariant S_{inv}, the proof search proceeds in four steps: (1) when we reach the loop, we abduct the invariant S_{inv} using **ex**; (2) we abduct the body of the loop assuming the loop invariant S_{inv}; (3) before the next iteration, we reestablish S_{inv} again using **ex**; finally, (4) we check that S_{inv} is indeed a loop invariant by ensuring the abduction of the loop body did not require any additional preconditions.

Failure. Finally, if no other Argon rule applies, the inference *fails* (ABD-FAIL). In this case, Argon terminates by inserting a marker into the precondition and provides the partial inferred precondition to the user. In impossible cases (*e.g.*, a location is `NULL`, but we are supposed to provide ownership), the marker can contain information explaining what went wrong, provided by the Argon instantiation. To remain sound, the marker is semantically interpreted as **False**, as indicated in ABD-FAIL.

4 THE TYPE SYSTEM THORIUM

In §2, we have seen how to apply abductive deductive verification to a simple separation logic. In Quiver, to scale to the complexities of C, we use a richer separation logic, **Thorium**. Following in the footsteps of RefinedC [52], Thorium is a separation logic-based type system. In the following, we explain its core ingredients: *type assignments* and *typed weakest preconditions*. A more detailed discussion of how Thorium builds proof search on top of these core ingredients—using *typing rules* in place of the weakest precondition rules in Fig. 4—is contained in the accompanying appendix [56].

Type assignments. Instead of abstract predicates $\mathrm{P}(v, x)$ and points-to assertions $\ell \mapsto v$ (as we considered in §2), resources in Thorium are *type assignments*. They are of the form $v \lhd_v A$ (read "v is

$$
\begin{array}{rrcl}
\text{Types} & A, B & ::= & \mathsf{void} \mid \mathsf{null} \mid \mathsf{num}[it]\, n \mid \mathsf{any}\, n \mid \mathsf{zeros}\, n \mid \mathsf{value}\, n\, v \mid \exists x.\, A\, x \mid A * P \\
& & \mid & \mathsf{own}\, \ell\, A \mid \mathsf{optional}\, \phi\, A \mid \mathsf{fn}\, T \mid \vec{x} \mathbin{@} \mathrm{P} \mid \mathsf{struct}[s]\, \vec{A} \mid \mathsf{array}[\mathrm{P}]\, xs \mid \cdots \\
\text{Resources} & M, N & ::= & v \triangleleft_v A \mid \ell \triangleleft_l A \mid \mathsf{block}\, \ell\, n \mid \cdots \\
\text{Embedded Goal} & E & ::= & \mathbf{wp}\, e\, \{v, A.\, \Phi\, v\, A\} \mid \mathbf{cast}\, (it_2)_{it_1}\, (v : A)\{w, B.\, \Phi\, w\, B\} \mid \cdots
\end{array}
$$

Fig. 9. Thorium types, resources, and goals.

an A") and $\ell \triangleleft_l A$ (read "ℓ stores an A"; semantically $(\exists v.\, \ell \mapsto v * v \triangleleft_v A) \vdash \ell \triangleleft_l A$). For each type A, they have an interpretation in terms of more traditional separation logic assertions. For example, the resource $v \triangleleft_v \mathsf{own}\, \ell\, (\mathtt{num[int]}\, n)$ means "v is an owned pointer ℓ storing the `int`-integer n", which formally boils down to $v \triangleleft_v \mathsf{own}\, \ell\, (\mathtt{num[int]}\, n) \Leftrightarrow v = \ell * \ell \mapsto n * n \in \mathtt{int}$.

The types of Thorium are depicted in Fig. 9. We explain the most important types by returning to the range data type (§2). In C, it would be declared as `typedef struct ran {int s; int e} *range;` and, for the predicate range$(v, n_\mathsf{s}, n_\mathsf{e})$ (from §2.3), the analogous Thorium type is defined as:

$$(n_\mathsf{s}, n_\mathsf{e}) \mathbin{@} \mathsf{range} \equiv_{\mathsf{ty}} \exists \ell.\, \mathsf{own}\, \ell\, (\mathsf{struct}[\mathtt{ran}]\, [A_\mathsf{s}; A_\mathsf{e}]) * 0 \le n_\mathsf{s} \le n_\mathsf{e} * \mathsf{block}\, \ell\, sz_\mathtt{ran}$$
$$\textit{where}\ \ A_\mathsf{s} \triangleq \mathtt{num[int]}\, n_\mathsf{s},\ A_\mathsf{e} \triangleq \mathtt{num[int]}\, n_\mathsf{e}, \textit{and}\ sz_\mathtt{ran} \triangleq \mathbf{sizeof}(\mathbf{struct}\ \mathtt{ran})$$

Types of the form $\vec{x} \mathbin{@} \mathrm{P}$ correspond to user-defined abstract predicates and are defined via a (possibly recursive) equation $\vec{x} \mathbin{@} \mathrm{P} \equiv_{\mathsf{ty}} A$. The type $(n_\mathsf{s}, n_\mathsf{e}) \mathbin{@} \mathsf{range}$ ensures that its values are owned pointers ℓ (via "own $\ell\, A$") to a `ran`-struct (via "$\mathsf{struct}[s]\, \vec{A}$") with two fields: `s` containing `int`-integer n_s (via "$\mathtt{num}[it]\, n$"), and `e` containing the `int`-integer n_e. To hide the location ℓ, we use type-level existential quantification "$\exists x.\, A\, x$" and, to impose the bounds constraint $0 \le n_\mathsf{s} \le n_\mathsf{e}$, we use type-level separating conjunction "$A * P$". Besides the bounds constraint, the type carries an additional constraint: the resource "block $\ell\, n$". It tracks the length of dynamically allocated blocks (*e.g.*, via `malloc`; see Fig. 1) to ensure that ownership of the entire block is given up when freeing ℓ.

Typed weakest preconditions. Instead of standard weakest preconditions $\mathbf{wp}\, e\, \{v.\, \Phi\, v\}$ (in §2), we use *typed weakest preconditions* $\mathbf{wp}\, e\, \{v, A.\, \Phi\, v\, A\}$ in Thorium: their postcondition Φ is about the resulting value v and, additionally, its type A. This type A can then be used by auxiliary embedded goals like $\mathbf{cast}\, (it_2)_{it_1}\, (v : A)\{\Phi\}$ (for C-level integer type cast) to steer the proof search—via *typing rules* in place of the weakest precondition rules in Fig. 4. For example, to cast an `int` into a `size_t` integer, the typing rule $\mathbf{cast}\, (\mathtt{size\_t})_{\mathtt{int}}\, (v : \mathtt{num[int]}\, n)\{\Phi\} \dashv n \ge 0 * \forall w.\, \Phi\, w\, (\mathtt{num[size\_t]}\, n)$ requires n to be non-negative and continues with n as a `size_t`-integer in the postcondition. We expand on how types and typing rules affect the proof search in the accompanying appendix [56].

5 IMPLEMENTATION

We have developed a prototype implementation of Quiver in Coq. More specifically, we have implemented the goal-directed abduction engine Argon $\Delta * [R] \vdash G$ (which embeds the typing rules of Thorium) as an *automated abduction procedure* in Coq. For a given C function (and possibly a sketch), it (1) infers a specification and, at the same time, (2) proves its correctness.

We use the Coq proof assistant as a foundation for Quiver for two main reasons: First, Quiver inherits Coq's rich logic for expressing complex correctness properties (as evaluated in §6). Second, it allows us to ensure the correctness of the inferred specifications. Concretely, we have proven Quiver's inference foundationally sound against RefinedC's C semantics, Caesium. Caesium provides a detailed formalization of C, modeling many challenging features ranging from bounded integers and pointer arithmetic, over uninitialized memory with poison semantics and address-of operator (also on local variables), to manipulation of the underlying byte-level representation of

```
1  vec_t mkvec(int n){ size_t s=sizeof(int)*(size_t)n; vec_t vec=xmalloc(sizeof(*vec));
2    vec->data=xzalloc(s); vec->len=n; [[q::type(? @vec_t)]] return vec; }
3
4  [[q::requires(^vec ◁ᵥ ? @vec_t)]][[q::ensures(^vec ◁ᵥ ? @vec_t)]]
5  int vec_grow(vec_t vec, int new_size)
6  { if (vec == NULL) { return 0; }
7    if (new_size <= vec->len) { return vec->len; }
8    int *buf = xmalloc(sizeof(int) * new_size);
9    memcpy(buf, vec->data, sizeof(int) * vec->len);
10   free(vec->data); vec->data = buf;
11   memset(&(vec->data[vec->len]), 0, sizeof(int) * (new_size - vec->len));
12   vec->len = new_size; return vec->len; }
```

Fig. 10. The implementation of the vector. Quiver annotations in blue.

values.[5] To prove Quiver sound against Caesium, we have used the separation logic framework Iris [29, 28, 30] to model Argon and Thorium. We have proven all rules sound against this model:

Theorem 5.1. *All Argon and Thorium rules are sound wrt. the Caesium C semantics.*

The automated abduction procedure combines the soundness of the individual rules into a foundational proof that the inferred specifications are sound. In our examples, we assume specifications for common operations from the C standard library (*e.g.*, `malloc`, `memset`, and `abort` in Fig. 1), which can be found in the accompanying appendix [56]. Thus,

Corollary 5.2. *Assuming the standard library function satisfy their specifications, the specifications inferred by Quiver are sound wrt. the Caesium C semantics.*

Finally, Quiver comes with a frontend that automatically translates annotated C code into (1) corresponding Caesium code, (2) type declarations in Thorium, and (3) calls to the abduction procedure for Argon. The abduction procedure is implemented using Coq's Ltac tactic language [16] and typeclass mechanism [55].

6 EVALUATION

To evaluate Quiver, we have applied it to several interesting case studies, listed in Fig. 12. We split our evaluation into two parts: First, we take a closer look at a specific case study, *a vector*, to get a sense of the kind of specifications that Quiver can infer (in §6.1). Then, we discuss the aggregate results of evaluating Quiver on these case studies (in §6.2).

6.1 The Vector Case Study

Inspired by C++ and Rust, a *vector* is a dynamically-sized array that tracks its length. An excerpt of the vector implementation is depicted in Fig. 10. In this implementation, vectors are of C type `typedef struct vector {int *data; int len;} *vec_t`. They are pointers to a struct with two fields: the `data`-field storing the contents of the vector in a dynamically allocated array of integers and the `len`-field tracking the length of the vector. For vectors in Quiver, we define the Thorium-data type:

$$xs \,@\, \mathsf{vec\_t} \equiv_{\mathsf{ty}} \exists \ell.\ \mathsf{own}\ \ell\ (\mathsf{struct}[\mathtt{vector}]\ [A_{\mathtt{data}}; A_{\mathtt{len}}]) * \mathsf{block}\ \ell\ sz_{\mathsf{vec}}$$

where $A_{\mathtt{data}} \triangleq \exists r.\mathsf{own}\ r\ (\mathsf{array}[\mathsf{num}[\mathtt{int}]]\ xs) * \mathsf{block}\ r\ (sz_{\mathtt{int}} \cdot \mathsf{len}\ xs)$, $A_{\mathtt{len}} \triangleq \mathsf{num}[\mathtt{int}]\ (\mathsf{len}\ xs)$

[5] Quiver's version of Caesium forgoes checking alignment of accesses as the resulting constraints would clutter the inferred specifications and we do not use the integer-pointer-casting semantics introduced by Lepigre et al. [34].

$$T_{\texttt{mkvec}}(v_{\texttt{n}})(\Phi) \triangleq \exists n.\ (v_{\texttt{n}} \triangleleft_v \texttt{num[int]}\ n * n \geq 0) * (\forall w.\ w \triangleleft_v 0^n\ @\ \texttt{vec\_t} \mathrel{-\!\!*} \Phi\, w)$$

$$T_{\texttt{grow}}(v_{\texttt{vec}}, v_{\texttt{new}})(\Phi) \triangleq \exists xs, n.\ (v_{\texttt{vec}} \triangleleft_v xs\ @\ \texttt{vec\_t} * v_{\texttt{new}} \triangleleft_v \texttt{num[int]}\ n) *$$
$$\textbf{if}\ n \leq \mathsf{len}\ xs\ \textbf{then}\ \forall w.\ v_{\texttt{vec}} \triangleleft_v xs\ @\ \texttt{vec\_t} * w \triangleleft_v \texttt{num[int]}\ (\mathsf{len}\ xs) \mathrel{-\!\!*} \Phi\, w$$
$$\textbf{else}\ \forall w.\ v_{\texttt{vec}} \triangleleft_v (xs +\!\!+ 0^{n-\mathsf{len}\ xs})\ @\ \texttt{vec\_t} * w \triangleleft_v \texttt{num[int]}\ n \mathrel{-\!\!*} \Phi\, w$$

Fig. 11. Inferred vector specifications, preconditions in light blue and postconditions in violet.

That is, for a mathematical list of integers xs, a value of type xs @ vec_t is an owned pointer ℓ to a `vector`-struct. It stores in its `data`-field an owned pointer r to an array of integers xs and in its `len`-field the length of xs as an integer. It tracks the memory block resources of ℓ of size $sz_{\texttt{vec}}$ and r of size $sz_{\texttt{int}} \cdot \mathsf{len}\ xs$ where $sz_{\texttt{vec}} \triangleq \textbf{sizeof}(\textbf{struct}\ \texttt{vector})$ and $sz_{\texttt{int}} \triangleq \textbf{sizeof}(\texttt{int})$.

We focus on two vector operations (see the appendix [56] for more): The operation `mkvec` creates a new vector of length n initialized with zeros, and the operation `vec_grow` extends a vector by allocating a new underlying buffer. Concretely, `vec_grow` allocates a new content array `buf` of larger size (Line 8), copies the contents of the old array over (Line 9), frees the old array (Line 10), sets all uninitialized memory to zero (Line 11), and returns the new length (Line 12).

Sketches and inferred specifications. For each operation, the *specification sketches* are annotated with "`[[q::...]]`" in Fig. 10, where **`q::requires`** sketches preconditions, **`q::ensures`** postconditions, **`q::type`** the type of an expression, and `^vec` means $v_{\texttt{vec}}$. The *inferred specifications* are depicted in Fig. 11. For `mkvec`, Quiver infers that the size n must be a non-negative `int`-integer and that the return value is a vec_t-vector filled with 0^n, a list of n zeros. For `vec_grow`, Quiver infers a conditional specification: if $n \leq \mathsf{len}\ xs$, the vector is unchanged and $\mathsf{len}\ xs$ is returned; otherwise, the vector grows by $n - \mathsf{len}\ xs$ zeros and n is returned as the new length. To arrive at this specification, Quiver (1) infers the type of the unspecified argument $v_{\texttt{new}}$, (2) resolves the quantifier alternations that arise from each memory operation (a $\exists\forall$ for each operation), (3) instantiates the sketches (including $xs +\!\!+ 0^{n-\mathsf{len}\ xs}$ @ vec_t for the second case), (4) proves that $\mathsf{len}(xs +\!\!+ 0^{n-\mathsf{len}\ xs}) = n$ when $n > \mathsf{len}\ xs$, and (5) prunes the branch returning `0` using the fact that xs @ vec_t is never `NULL`.

Abductive deductive verification. The vector case study illustrates concisely the benefits of abductive deductive verification. On the one hand, we are doing expressive separation logic verification. For example, (a) vectors track their contents as a mathematical list of integers, (b) vectors maintain the invariant that the length of the list is stored in the field `len`, (c) dynamically allocated memory can be of variable length, which is tracked via a predicate block, (d) pointer arithmetic is used to compute fields of structs and members of arrays, and (e) pointer-level operations (*e.g.*, `memset` and `memcpy`) are used to manipulate high-level data types (*e.g.*, arrays). On the other hand, we can significantly benefit from inference for the verification. In particular, we only need to provide the key bit of information—that a certain value is a vector—and can use inference to complete the rest. In the accompanying appendix [56], we show that the same sketches suffice for additional, quite different vector operations (*e.g.*, for getting and setting elements in the vector).

6.2 Aggregate Evaluation

We evaluate the prototype implementation of Quiver on three axes: (1) the expressivity (compared to bi-abduction), (2) the specification overhead (compared to RefinedC), and (3) the merit of the inferred specifications. We do so using the case studies in Fig. 12. For each case study, we provide a more detailed discussion in the appendix and all implementations and inferred specifications can

Implementation			Specification					Execution		
Name	Functions	Code	Type	Specs	Sketch	Annot	Coq	∃	ϕ	Time
Allocators	`xmalloc`, `xzalloc`, `xrealloc`, …(+3)	41	mem	55	0	0	0	77	44/31	0:58
Linked list	`init`, `is_empty`, `push`, `pop`, `reverse` (only functional)	37	mem	27	0	0	0	16	11/2	0:27
			func	25	10	11/5/0	0	39	21/8	0:46
Vector	`mkvec`, `get_unsafe`, `grow`, `get_checked`, `vec_free`, …(+3)	59	mem	147	0	0	0	106	62/27	2:48
			func	75	14	11/0/0	0	117	164/43	2:40
Bipbuffer	`new`, `free`, `request`, `push`, …(+11)	105	len	210	21	10/0/2	0	378	476/160	8:51
OpenSSL Buffer	`BUF_MEM_new`, `BUF_MEM_free`, `BUF_MEM_grow`, …(+3)	107	mem	249	0	0	0	285	302/113	14:14
			len	94	9	14/0/4	0	310	431/90	9:50
Binary search	`bin_search`	14	func	11	5	7/8/0	49	18	49/3	0:41
Hashmap	`init`, `probe`, `realloc`, …(+5)	101	func	79	72	19/18/7	506	221	375/123	7:56

Fig. 12. Evaluation of Quiver. Code: lines of C code as formatted by clang-format; Type: type of inferred specification (*i.e.*, mem: memory safety, len: length and type invariants, func: functional); Specs: size of the inferred specification; Sketch: size of the function sketches; Annot: size of type definitions/size of loop invariants/additional inference instructions; Coq: pure Coq definitions and lemmas; "∃": number of instantiated existential quantifiers; "ϕ": number of proven/simplified side conditions; Time: execution time in minutes:seconds.

be found in the Coq development [56]. The Allocators case study considers common wrappers around standard library functions for memory allocation (*e.g.*, `xmalloc` and `xzalloc`). The Linked List case study considers a singly linked-list implementation with pointer elements, and the Vector case study extends the vector from §6.1. The OpenSSL Buffer and Bipbuffer case studies consider open-source buffer implementations from OpenSSL [42] and memcached [37]. The Binary Search case study considers binary search on sorted integer lists, and the Hashmap case study considers a hashmap with linear probing. For each case study, we measure the execution time on a single core of an Apple M1 Pro processor (Time).

Expressivity (*vs.* Bi-abduction). To understand the degree of expressivity that Quiver supports, we consider several types of specifications (Type in Fig. 12), increasing in complexity: We infer *memory safety specifications* (mem) for several examples—including the Allocators, whose inferred specifications (*e.g.*, `xmalloc` and `xzalloc`) we use in other case studies. We infer *length specifications* (len) for the open-source buffers, which track the length of the buffer and data type invariants about its fields. We infer *functional specifications* (func) for the Linked List and the Vector, which track their contents as mathematical lists. And, to test the boundaries of Quiver, we consider a binary search implementation and a Hashmap, a version of the most complex functional correctness case study of Sammler et al. [52] specialized to integer values.[6]

The case studies demonstrate that Quiver, embedded into Coq, supports expressive separation logic reasoning over a variety of mathematical domains (*e.g.*, integers, lists, maps, and custom inductive types). For example, Quiver figures out that (a) if $n < \texttt{0x5ffffffc}$, then $(n + 3)/3 \cdot 4$ will not overflow the `size_t` type (OpenSSL Buffer) and (b) `grow` results in the vector $xs +\!\!+ 0^{n - \operatorname{len} xs}$, which extends the original list xs with $n - \operatorname{len} xs$ zeros (Vector). Moreover, provided with loop invariants and additional Coq lemmas and definitions, Quiver does significant functional correctness reasoning for the Binary Search and Hashmap. The expressivity of Quiver goes considerably beyond

[6]Differences to the implementation of Sammler et al. [52] are discussed in the supplementary material.

the original bi-abduction inference [5, 6] and also what is nowadays used in Infer [26].[7] In exchange, it requires more input from the user, in particular for more expressive specifications.

Specification overhead (*vs.* RefinedC). To understand how much specification Quiver infers, we compare the size of the inferred specifications (Specs) with the size of our sketches (Sketch) and other annotations (Annot). We measure the size of specifications and sketches by counting the number of quantifiers, conditionals, conjunctions, type assignments, and other individual pre- and postconditions (*e.g.*, the size of $T_{\texttt{mkvec}}$ would be 5). We separately count other annotations such as type definitions, loop invariants, and inference instructions. A handcrafted specification—as it would be provided in RefinedC—could in some cases reduce the size (*e.g.*, by joining the branches in $T_{\texttt{grow}}$), but nevertheless comparing sketches and specs gives an idea how much Quiver infers. Concretely, for the "memory" case studies, we provide no sketches—the specifications are completely inferred. By design, they are low-level (*e.g.*, see Fig. 1) and can be verbose. For all other case studies, we provide sketches. They are typically significantly smaller than the resulting specification and often contain `?`-holes (*e.g.*, all 14 Vector sketches boil down to `? @ vec_t`). In RefinedC, by contrast, specifications have to be provided in full. Among our case studies, there are two outliers: the Binary Search and the Hashmap. This is no surprise, since both require nuanced, ad-hoc functional correctness reasoning with additional pure Coq definitions and lemmas (Coq). For them, the specification overhead is overshadowed by the additional proof overhead. Nevertheless, even for those two, Quiver does interesting inference: it completes the return type of Hashmap `init`, and it derives the postcondition of the Binary Search from a loop invariant.

Merit of the specifications. The specifications that Quiver infers provide four key benefits: First, they are an additional form of *documentation*. Quiver outputs a pretty-printed version of the inferred predicate transformer, which can be read by humans. For example, in the Vector, Quiver adds the constraints on the vector size in the specification of `mkvec`. Second, the inferred specifications provide *assurances* about the code. That is, due to soundness (Corollary 5.2), the inferred specifications cannot "hide" any preconditions that are undocumented in the code. For example, in the Bipbuffer, Quiver discovers a fact about the implementation that is easy to miss in the code: the implementation uses mismatched integer types (*e.g.*, the size field of the buffer uses `unsigned long int`, but the corresponding accessor function returns `int`), resulting in an additional precondition in the generated specifications. Third, the inferred specifications are *compositional*. We inherit compositionality from working in separation logic. In particular, in many of the case studies, we infer specifications of auxiliary functions, which are then reused in the inference of others (*e.g.*, BipBuffer, OpenSSL Buffer, and Hashmap); and we use the inferred Allocator specifications in other case studies (*e.g.*, in the Vector, List, Hashmap). Fourth, the inferred specifications *abstract over the implementation*. By insisting on a single $\exists\forall$-alternation, Quiver ensures that the inferred specification condenses the implementation into preconditions and postconditions. In doing so, it takes care of the intricacies of the C implementation and intermediate proof obligations. To gain some insight into how much work goes into this summarization, we count the number of instantiated existentials ($\exists$) and proven/simplified side conditions (ϕ).

Real-world code. Finally, our case studies test whether Quiver can handle the complexities of real-world code. We have applied Quiver to two buffer implementations taken from popular open source libraries, OpenSSL [42] and memcached [37]. For the OpenSSL buffer, we track the length and capacity of the buffer and enforce an invariant that the buffer capacity is always larger than

[7] For example, Infer does not do integer reasoning such as (a) if $n < \texttt{0x5ffffffc}$, then $(n+3)/3 \cdot 4 \leq \texttt{0xffffffff}$ from the OpenSSL buffer or (b) after the loop `int k = 0; while (k < 10) k++` the counter k is 10. Quiver automatically proves the former without any guidance, and infers the latter when guided with the loop invariant $k \leq 10$.

the contents. For the memcached buffer, a *bipartite buffer*, we track the length and the relationship between the fields that track the segments of the buffer.

7 RELATED WORK

In the literature on separation logic verification, there is a wide gap between approaches for (a) automatically inferring specifications *vs.* (b) verifying functional correctness in rich separation logics. In the first camp, there are approaches such as bi-abduction [5, 6], which fix a particular fragment of separation logic, and then carefully design automation to infer specifications in it. This line of work started out with shape specifications (*i.e.*, linked list segments and points-to assertions) and, over the years, edged closer toward functional properties by extending the base domain to include constraints on integers, arrays, or bags. In the second camp, there are approaches such as RefinedC [52], which are designed for proving full functional correctness in rich separation logics, as supported by the verification frameworks in which they are embedded (*e.g.*, Coq [13] and Iris [29]). This line of work, over the years, developed increasingly strong proof automation but left specification inference largely untouched.

Quiver sits right in between these two camps, supporting a wide range in between automated and expressive specifications (see §6.2). Typically, Quiver requires more specification guidance from users than a fully automatic inference (increasing with expressiveness of the specification), but significantly less than traditional, deductive approaches for rich separation logics. In exchange, it does not fix any particular mathematical domain and, instead, is implemented in a general-purpose proof assistant—producing certifiably correct specifications. We first compare closely with work in both camps, and then branch out to other related work.

Inferring ownership specifications in separation logic. In their seminal work, Calcagno et al. [5, 6] introduced *bi-abduction* as a technique for compositional shape analysis in separation logic. Over the years, extensions to its original domain (*i.e.*, points-tos and list segments) have been proposed, including pure constraints over booleans, integers, and bags [57, 46, 23]; ordering constraints [14]; low-level data representations [25]; second-order predicates [31]; and arrays [3]. The key contribution of these extensions is to automate the inference over their respective domain.

In contrast, Quiver's specification inference is fundamentally different. By using abductive deductive verification, Quiver is less automated but, in exchange, handles a much richer separation logic by building on existing approaches for deductive proof automation. For example, the vector example (§6.1)—combining low-level pointer operations, arrays, and integer arithmetic—goes beyond all of these extensions, especially considering the detailed C semantics it is verified against.

Outside of the context of bi-abduction, Dohrau et al. [19] use a static analysis to infer access permissions for array-manipulating programs, and Ferrara and Müller [20] show how to automatically infer access permissions using abstract interpretation. They handle different permission models and loop invariant inference but do not consider functional correctness properties.

Functional correctness verification using separation logic. There is a wide range of approaches for verifying functional correctness based on separation logic [27, 2, 7, 39, 52, 45], most of which do not infer specifications. We compare to the most closely related work and approaches with some form of specification inference.

A key inspiration for Quiver is RefinedC [52], which provides automated and foundational verification of C code. Its approach of using a type system embedded in separation logic served as a direct inspiration for Quiver. However, RefinedC does not infer specifications and, hence, relies on *user-provided, complete specifications*. To tackle specification inference, we introduced the abductive deductive verification approach, implemented a proof engine for abduction—Argon—from scratch, and designed a type system—Thorium—that integrates seamlessly with abduction.

For VeriFast [27], a separation logic-based functional correctness verifier for C and Java, Vogels et al. [59] implement a bi-abduction-based shape analysis. Unlike Quiver, it does not infer functional correctness specifications and only infers a postcondition from a user-provided precondition. Separately, Automated VeriFast [38] leverages errors reported by VeriFast to extend user-written specs with additional pre- and postconditions. Automated VeriFast has only been demonstrated on predicates tracking the length of singly-linked lists.

Dohrau [18] presents a learning-based permission inference for the Viper automated verifier [39]. Their approach can automatically infer loop invariants and predicate definitions, but only considers permissions, not functional correctness properties.

Liquid types. Liquid types [50, 51, 58, 33] provide a refinement type-based approach for light-weight verification. Liquid types focus on the inference of pure refinements, not separation logic ownership, and often consider more shape-like properties than Quiver. For example, Lehmann et al. [33] describe a vector similar to vec_t from §2, but only track the length in the refinements, not its precise contents. In exchange, liquid types are more automated: they infer refinements and, additionally, loop invariants automatically.

Specification inference for other logics. Outside of the context of separation logic, a separate body of research [54, 1, 44] considers inferring specifications for programs that do not involve pointer manipulation or a heap. This restriction sidesteps the main challenges this paper focuses on (see, *e.g.*, the vector in §6.1). In exchange, they typically obtain *exact* (*i.e.*, sufficient and necessary) preconditions, whereas Quiver infers sufficient preconditions.

Characteristic formulae. A characteristic formula [10, 11] is a direct translation of a program into a separation logic formula. Characteristic formulae are not intended as specifications, but as an intermediate representation used during verification. In particular, they still contain all intermediate proof obligations required to verify a function. In contrast, Quiver infers specifications that *summarize* the behavior of a function in terms of pre- and postconditions (*i.e.*, in $\exists\forall$-form; see §2.2) by resolving quantifier dependencies and solving side conditions.

8 CONCLUSION AND FUTURE WORK

With Quiver, we have introduced the idea of abductive deductive verification (§2) and applied it to specification inference of C programs, using the abduction engine Argon (§3) and the separation logic based type system Thorium (§4). In the future, it would be interesting to apply abductive deductive verification, and in particular Argon, to other languages and flavors of separation logic.

Moreover, it would be interesting to investigate *loop invariant inference*. Finding loop invariants in separation logic is a non-trivial task: it requires finding pure invariants and, additionally, invariants about resources. For *restricted fragments* of separation logic, loop invariant inference techniques have been developed [35, 5, 23, 25]. But for rich separation logics like the one targeted by Quiver, no loop invariant inference algorithms are known. Thus, like Quiver, deductive verification tools for expressive separation logics (*e.g.*, VeriFast, CN, Viper, and RefinedC) require user-provided loop invariants. It would be interesting to investigate how to integrate (a) existing loop invariant inference algorithms for separation logic when the invariant falls into a supported fragment, (b) learning techniques as an approach to loop invariant inference, or (c) existing *non-separation logic* loop invariant inference techniques by requiring loop invariant sketches (for the resources) but leaving holes for the pure invariants—potentially using abduction to determine how the pure values evolve inside the loop.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This work was funded in part by Google PhD Fellowships for the first and third authors.

DATA AVAILABILITY STATEMENT

The Coq development and appendix for this paper can be found in Spies et al. [56]. The current development version of Quiver is linked from the project webpage at https://plv.mpi-sws.org/quiver/.

REFERENCES

[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*. ACM, 789–801. https://doi.org/10.1145/2837614.2837628
[2] Andrew W. Appel. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. https://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers
[3] James Brotherston, Nikos Gorogiannis, and Max I. Kanovich. 2017. Biabduction (and Related Problems) in Array Separation Logic. In *CADE (LNCS, Vol. 10395)*. Springer, 472–490. https://doi.org/10.1007/978-3-319-63046-5_29
[4] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2019. *Go Huge or Go Home: POPL'19 Most Influential Paper Retrospective*. https://blog.sigplan.org/2020/03/03/go-huge-or-go-home-popl19-most-influential-paper-retrospective/
[5] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*. ACM, 289–300. https://doi.org/10.1145/1480881.1480917
[6] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700
[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5
[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. ACM, 243–258. https://doi.org/10.1145/3341301.3359632
[9] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *OSDI*. USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed
[10] Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *ICFP*. ACM, 321–332. https://doi.org/10.1145/1863543.1863590
[11] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *ICFP*. ACM, 418–430. https://doi.org/10.1145/2034773.2034828
[12] Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ICFP*. ACM, 391–402. https://doi.org/10.1145/2500365.2500592
[13] Coq. 2023. The Coq proof assistant. https://coq.inria.fr/.
[14] Christopher Curry, Quang Loc Le, and Shengchao Qin. 2019. Bi-Abductive Inference for Shape and Ordering Properties. In *ICECCS*. IEEE, 220–225. https://doi.org/10.1109/ICECCS.2019.00031
[15] Cyrus IMAPD. 2023. Cyrus IMAPD Memory Wrapper Operations. https://github.com/cyrusimap/cyrus-imapd/blob/0552750789f23d205b50f582f73358d73cc15706/lib/xmalloc.c.
[16] David Delahaye. 2000. A Tactic Language for the System Coq. In *LPAR (LNCS, Vol. 1955)*. Springer, 85–95. https://doi.org/10.1007/3-540-44404-1_7
[17] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975
[18] Jérôme Dohrau. 2022. *Automatic Inference of Permission Specifications*. Ph. D. Dissertation. ETH Zurich, Zürich, Switzerland. https://doi.org/10.3929/ethz-b-000588977
[19] Jérôme Dohrau, Alexander J. Summers, Caterina Urban, Severin Münger, and Peter Müller. 2018. Permission Inference for Array Programs. In *CAV (2) (LNCS, Vol. 10982)*. Springer, 55–74. https://doi.org/10.1007/978-3-319-96142-2_7
[20] Pietro Ferrara and Peter Müller. 2012. Automatic Inference of Access Permissions. In *VMCAI (LNCS, Vol. 7148)*. Springer, 202–218. https://doi.org/10.1007/978-3-642-27940-9_14
[21] Robert W Floyd. 1967. Assigning meanings to programs. *American Mathematical Society* (1967).

[22] Git. 2023. Git Memory Wrapper Operations. https://github.com/git/git/blob/2e8e77cbac8ac17f94eee2087187fa1718e38b14/wrapper.c.
[23] Guanhua He, Shengchao Qin, Wei-Ngan Chin, and Florin Craciun. 2013. Automated Specification Discovery via User-Defined Predicates. In *ICFEM (LNCS, Vol. 8144)*. Springer, 397–414. https://doi.org/10.1007/978-3-642-41202-8_26
[24] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259
[25] Lukás Holík, Petr Peringer, Adam Rogalewicz, Veronika Soková, Tomás Vojnar, and Florian Zuleger. 2022. Low-Level Bi-Abduction. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:30. https://doi.org/10.4230/LIPIcs.ECOOP.2022.19
[26] Infer. 2023. Infer. https://fbinfer.com.
[27] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS, Vol. 6617)*. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4
[28] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151
[29] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980
[30] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855
[31] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. 2014. Shape Analysis via Second-Order Bi-Abduction. In *CAV (LNCS, Vol. 8559)*. Springer, 52–68. https://doi.org/10.1007/978-3-319-08867-9_4
[32] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527325
[33] Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. (2023). https://doi.org/10.1145/3591283
[34] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–32. https://doi.org/10.1145/3498681
[35] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring invariants in separation logic for imperative list-processing programs. *SPACE* 1, 1 (2006), 5–7.
[36] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38
[37] memcached. 2023. memcached. https://www.memcached.org/.
[38] Mahmoud Mohsen and Bart Jacobs. 2016. One Step Towards Automatic Inference of Formal Specifications Using Automated VeriFast. In *FMICS-AVoCS (LNCS, Vol. 9933)*. Springer, 56–64. https://doi.org/10.1007/978-3-319-45943-1_4
[39] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2
[40] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078
[41] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1
[42] OpenSSL. 2023. OpenSSL. https://www.openssl.org.
[43] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (LNCS, Vol. 8413)*. Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9
[44] Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification synthesis with constrained Horn clauses. In *PLDI*. ACM, 1203–1217. https://doi.org/10.1145/3453483.3454104
[45] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. https://doi.org/10.1145/3571194
[46] Shengchao Qin, Chenguang Luo, Wei-Ngan Chin, and Guanhua He. 2011. Automatically Refining Partial Specifications for Program Verification. In *FM (LNCS, Vol. 6664)*. Springer, 369–385. https://doi.org/10.1007/978-3-642-21437-0_28

[47] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *CAV (2) (LNCS, Vol. 12225)*. Springer, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

[48] Redis. 2023. Redis Memory Wrapper Operations. https://github.com/redis/redis/blob/3fac869f02657d94dc89fab23acb8ef188889c96/src/zmalloc.c.

[49] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[50] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. ACM, 159–169. https://doi.org/10.1145/1375581.1375602

[51] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL*. ACM, 131–144. https://doi.org/10.1145/1706299.1706316

[52] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[53] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *PLDI*. ACM, 927–942. https://doi.org/10.1145/3385412.3386014

[54] Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *ESOP (LNCS, Vol. 7792)*. Springer, 451–471. https://doi.org/10.1007/978-3-642-37036-6_25

[55] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS, Vol. 5170)*. Springer, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[56] Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. 2024. Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq (Coq development and Appendix). https://doi.org/10.5281/zenodo.10940320 Project webpage with appendix: https://plv.mpi-sws.org/quiver/.

[57] Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. 2013. Bi-Abduction with Pure Properties for Specification Inference. In *APLAS (LNCS, Vol. 8301)*. Springer, 107–123. https://doi.org/10.1007/978-3-319-03542-0_8

[58] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282. https://doi.org/10.1145/2628136.2628161

[59] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation Inference for Separation Logic Based Verifiers. In *FMOODS/FORTE (LNCS, Vol. 6722)*. Springer, 319–333. https://doi.org/10.1007/978-3-642-21461-5_21

Received 2023-11-16; accepted 2024-03-31