A Logical Approach to Type Soundness

AMIN TIMANY, Aarhus University, Denmark
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEREK DREYER, MPI-SWS, Germany
LARS BIRKEDAL, Aarhus University, Denmark

Type soundness, which asserts that "well-typed programs cannot go wrong", is widely viewed as the canonical theorem one must prove to establish that a type system is doing its job. It is commonly proved using the so-called *syntactic approach* (aka *progress and preservation*), which has had a huge impact on the study and teaching of programming language foundations. Unfortunately, syntactic type soundness is a rather weak theorem. It only applies to programs that are completely well-typed, and thus tells us nothing about the many programs written in "safe" languages that make use of "unsafe" language features. Even worse, it tells us nothing about whether type systems achieve one of their main goals: enforcement of data abstraction. One can easily define a language that enjoys syntactic type soundness and yet fails to support even the most basic modular reasoning principles for abstraction mechanisms like closures, objects, and abstract data types.

In this paper, we argue that we should no longer be satisfied with just proving syntactic type soundness, and should instead start proving a stronger theorem—*semantic type soundness*—which captures more accurately what type systems are actually good for. Semantic type soundness is an old idea—Milner's original formulation of type soundness was a semantic one—but it fell out of favor in the 1990s due to limitations and complexities of denotational models. In the succeeding decades, thanks to a series of technical advances—notably, (1) *step-indexed Kripke logical relations* constructed over operational semantics and (2) *higher-order concurrent separation logic* as consolidated in the *Iris* framework in Coq—we can now build (machine-checked) semantic soundness proofs at a much higher level of abstraction than was previously possible.

The resulting "logical" approach to semantic type soundness has already been employed to great effect in a number of recent papers (by us and others), but those papers typically concern advanced problem scenarios that complicate the presentation, they assume significant prior knowledge of the reader, and they refrain from giving many details of the proofs. Here, we hope to provide a gentler, more pedagogically motivated introduction to logical type soundness, aimed at a broader audience that may or may not be familiar with logical relations and Iris. As a bonus, we also show how logical type soundness proofs can be easily generalized to establish an even stronger *relational* property—*representation independence*—for realistic type systems.

> Type structure is a syntactic discipline for enforcing levels of abstraction.
> – Reynolds [1983]
>
> Although types and assertions may be semantically similar, the actual development of type systems for programming languages has been quite separate from the development of approaches to specification such as Hoare logic. . . the real question is whether the dividing line between types and assertions can be erased. – Reynolds [2002]

This paper is dedicated to the memory of John C. Reynolds.

1 INTRODUCTION

The *type soundness* (or *type safety*) theorem for a programming language states that if a program in that language passes the type checker, it should be guaranteed to have well-defined behavior when executed. Introduced over 40 years ago by Milner [1978], type soundness has become the canonical property that type systems for "safe" programming languages are expected to satisfy.

Authors' addresses: Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

In Milner [1978]'s original formulation for a λ-calculus with ML-style polymorphism, type soundness was characterized using denotational semantics. Ill-behaved programs were assigned a special denotation "wrong", and the type soundness theorem stated that well-typed programs could not "go wrong" (*i.e.,* have "wrong" as their denotation). However, it turned out to be difficult to scale this methodology to richer type systems with features such as general recursive types, higher-order mutable state, control operators, and concurrency.

Syntactic type soundness. Today, the most common formulation of type soundness is the so-called "syntactic approach", pioneered by Wright and Felleisen [1994] and subsequently simplified by Harper [2016] into the two theorems known as "progress and preservation".[1] Instead of employing a "wrong" denotation, the syntactic approach characterizes undefined behavior *operationally*: a program has undefined behavior if its execution under a small-step operational semantics "gets stuck" (*i.e.,* reaches a non-terminal state where there is no next step of execution to take). The preservation theorem states that a program remains well-typed as it executes, and the progress theorem states that a well-typed program is either in a terminal state or its next step of execution is well-defined.[2] Together, these theorems imply that the execution of a well-typed program is well-defined in the sense that it never gets stuck.

The syntactic approach to type soundness via progress and preservation is arguably one of the "greatest hits" of programming languages research of the past three decades. In addition to being conceptually simple, the approach scales easily to handle a wide range of programming language features, and has been popularized effectively through the central organizing role it plays in the textbooks of Pierce [2002] and Harper [2016]. As a result, it has become one of the most widely known, widely taught, and widely applied formal methods in the entire area of programming language foundations, with countless research papers on type systems concluding triumphantly with a statement of progress and preservation.

The limitations of syntactic type soundness. Unfortunately, syntactic type soundness also suffers from two significant limitations that are not (in our experience) widely recognized.

The first limitation pertains to *data abstraction.* One of the primary functions of the type systems of many languages is to give programmers a way of enforcing data abstraction boundaries, so that one can place invariants on the private data representations of modules, objects, abstract data types, *etc.* and be sure that client code will not violate those invariants. However, syntactic type soundness offers no guarantees about whether a programming language's data abstraction facility actually works—it is extremely easy to prove syntactic soundness of a type system whose data abstraction mechanism is completely broken.

The second limitation pertains to *unsafe features.* In practice, most "safe" languages provide unsafe escape hatches—*e.g.,* `Obj.magic` in OCaml, `unsafePerformIO` in Haskell, `unsafe` blocks in Rust—which enable programmers to perform potentially unsafe operations (such as unchecked type casts or array accesses) that the safe fragment of the language disallows. These unsafe escape hatches have proven indispensable, both for functionality—when the language's safe type system is more restrictive than necessary—and for efficiency—when performance concerns demand the use of a lower-level unsafe abstraction. However, syntactic type soundness has nothing to say about programs that use unsafe features: it simply declares such programs out of scope.

These two limitations are in fact closely connected, in that it is common to justify the "safe" use of unsafe features by appeal to data abstraction. Specifically, programmers often argue informally

[1] Harper is responsible for suggesting the progress theorem (and the associated "canonical forms" lemma), which superseded Wright and Felleisen's more complex analysis of "faulty" expressions.

[2] In order to state these theorems, one must first generalize the notion of well-typed programs to a notion of well-typed machine states, but this is typically straightforward.

that their use of unsafe operations is harmless because said operations have been *encapsulated* behind a "safe API". That is, they argue that, thanks to the abstraction boundary of the API, the implementation of the API can enforce invariants on its private data representation which ensure that its use of unsafe features does not lead to any undefined behavior. But of course, to make this reasoning formal, one needs to know whether the language is enforcing data abstraction properly, precisely one of the issues on which syntactic type soundness is silent.

Together, these limitations suggest that syntactic type soundness does not provide a sufficient foundation for judging whether a type system is really doing its job. So one may wonder: can we do any better? And in this article, we argue: yes, we can!

Logical type soundness. We propose an alternative to syntactic type soundness that overcomes the aforementioned limitations, offering a flexible foundation for reasoning about data abstraction, as well as the safe use of unsafe features. We call our approach *logical type soundness*. The essence of logical type soundness is not new: it is the age-old idea of *semantic type soundness*, as exemplified by the formulation in Milner [1978]'s original paper. Under the semantic soundness approach, one defines a semantic model of types, which offers an extensional view of typing rather than an intensional one. In other words, unlike syntactic typing, which dictates the syntactic structure of well-typed terms, semantic typing merely places restrictions on their observable behavior. As such, it enables us to explain when a term behaves safely at a given type, even if the term employs unsafe or lower-level operations internally.

Although Milner built his semantic model over a denotational semantics, we follow more recent approaches [Birkedal et al. 2011; Schwinghammer et al. 2013] and build ours over an operational semantics. In particular, we are inspired by the work by Appel, Ahmed, and collaborators on the Foundational Proof-Carrying Code project [Appel and Felty 2000; Appel 2001; Appel and McAllester 2001; Ahmed et al. 2002; Ahmed 2004; Ahmed et al. 2010], which demonstrated how to scale semantic soundness to account for a wide variety of programming language features using the powerful technique of *step-indexed models*.

The key point of difference between our approach and theirs is the level of abstraction at which the semantic soundness proof is conducted. As we explain in detail in §4.3 (and as previously noted by Appel et al. [2007] and Dreyer et al. [2011]), prior work that built semantic soundness proofs directly using step-indexed models involved a great deal of explicit reasoning about step-indexing and about the quasi-circular constructions that step-indexing served to disentangle. Such reasoning quickly became very tedious, to the point that the high-level structure of a proof would often become obscured if one were to write out all the low-level details.

In contrast, we show how to lift semantic soundness proofs to a much higher level of abstraction by employing recent advances in *higher-order concurrent separation logic* [Svendsen et al. 2013; Svendsen and Birkedal 2014]—hence the name "*logical* type soundness". Specifically, we show how by using the separation-logic framework *Iris* [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b], we can formulate semantic soundness proofs—*for feature-rich, realistic languages*—in a clear and concise manner, uncluttered by the low-level details of prior accounts. As a major added bonus, Iris is implemented in the Coq proof assistant and provides effective tactical support for constructing machine-checked logical type soundness proofs with relative ease [Krebbers et al. 2017b, 2018].

Type soundness expresses a property of a single program and hence it is sometimes referred to as a *unary* property. It turns out that our logical approach to type soundness can be easily adapted to support *relational reasoning* as well. Here, relational reasoning refers to properties about *pairs* of programs, *i.e.,* properties that are sometimes also referred to as *binary* properties. A particularly important example of such a binary property is *representation independence* [Reynolds 1974; Mitchell

1986]. Representation independence is a strong guarantee on the effectiveness of a language's data abstraction facility, even stronger than semantic soundness—it ensures that one can change the internal data representation of an *abstract data type* (*ADT*)[3] without affecting the behavior of its clients. Yet similarly to semantic type soundness, prior state-of-the-art semantic models for proving representation independence have typically been expressed directly in set theory using explicit step-indexing, see *e.g.*, Neis et al. [2009]; Ahmed et al. [2009]; Dreyer et al. [2010]; Thamsborg and Birkedal [2011]; Dreyer et al. [2012]; Birkedal et al. [2012, 2013]. We demonstrate by example how the advanced features of Iris can be used to formalize (machine-checked) proofs of representation independence at a higher level of abstraction than was previously possible.

Goal of this paper. Over the last five years, many papers have demonstrated that the logical approach to type soundness in Iris is eminently practical and scalable: among other things, it has been used for a machine-checked proof of type soundness of a significant subset of the Rust programming language [Jung et al. 2018a, 2021; Jung 2020; Dang et al. 2020], an extension of Scala's core type system DOT [Giarrusso et al. 2020], session types [Hinrichsen et al. 2021], and refinement types for the C programming language [Sammler et al. 2021]. Aside from type soundness, the logical approach has also been used to prove robust safety [Swasey et al. 2017; Sammler et al. 2020], various forms of representation independence and program refinement [Krogh-Jespersen et al. 2017; Tassarotti et al. 2017; Timany et al. 2018; Timany and Birkedal 2019; Frumin et al. 2018, 2021b; Jacobs et al. 2021], and various security properties [Frumin et al. 2021a; Gregersen et al. 2021; Georges et al. 2021].

The aforementioned papers are driven by particular applications and thus use the logical approach in sophisticated ways, typically in the context of a complicated programming language, type system, or program property. As a consequence, those papers typically omit many details and presuppose expert knowledge. Our goal here in this paper is instead pedagogical: we aim to make the general technique of logical type soundness better known to a wider audience. Thus, we present it in the context of a simple programming language with a fairly pedestrian set of features and without assuming that the reader is already well-versed in separation logic and step-indexing. Our intention is that this paper should be accessible to researchers and (under)graduate students who are familiar with basic textbooks in programming language theory such as Pierce [2002] or Harper [2016].

A note about the proofs in this paper: Most of our proofs are carried out *within* the Iris logic, and are thus of a rather different (and likely unfamiliar) nature compared with proofs in ordinary (higher-order) logic. Hence, we use proof trees to spell out our Iris proofs in great detail, and to show exactly which proof rules are applied where. However, we hasten to note that this is merely for formal clarity; in practice, when you are developing such Iris proofs *in Coq* (as nearly all Iris users do), much of this explicit detail is kept implicit, since the Iris Proof Mode [Krebbers et al. 2017b, 2018] keeps track of the Iris proof context and performs many "boring" proof steps for you *automatically*. Though a presentation of the Iris Proof Mode is beyond the scope of this paper, we refer the interested reader to the above-cited papers and accompanying online tutorials (see §10) for further details.

Outline. In §2, we define a simple, representative, feature-rich programming language—with higher-order state, recursive types, abstract types, and concurrency—which we will use throughout the rest of the paper, and we sketch the syntactic type soundness result for it. In §3, we explain the limitations of syntactic type soundness in more detail. In §4, we give a high-level description of the logical approach to type soundness, and provide an extensive comparison of our approach to prior work on semantic type soundness. In §5, we present the definition of a *logical relation*—the

[3] The acronym ADT is sometimes used to mean "*algebraic* data type", but in this paper we always mean "abstract data type".

core ingredient for proving logical type soundness in Iris—and in §6, we present the corresponding proofs. In §7, we show how the logical approach allows us to reason about safe encapsulation of unsafe features, and in §8, we extend the logical approach to support relational reasoning about representation independence. The relevant features and proof rules of Iris are introduced along the way in §5–§8. Finally, in §9, we discuss related work, and in §10 we conclude with a brief discussion of recent work that has employed our logical approach to type soundness and relational reasoning.

Origin of this paper. The technical content of this paper is based in part on Krebbers et al. [2017b, §6] and Timany [2018, Chapter 5]. Krebbers et al. [2017b, §6] provide a (2-page) case study showing that Iris and the Iris Proof Mode can be used for the mechanization of both semantic type soundness and representation independence proofs, and Chapter 5 of Timany [2018]'s PhD thesis provides a more extensive description of this case study. The present paper can be seen as a significant expansion of the above, explaining semantic type soundness from first principles in a more didactic fashion, without requiring prior knowledge of Iris, and with motivating examples drawn from the keynote talk that Dreyer gave at the POPL 2018 conference [Dreyer 2018].

2 THE LANGUAGE MYLANG AND ITS SYNTACTIC TYPE SOUNDNESS

We present the syntax and the semantics of our subject of study: the language **MyLang**—a call-by-value λ-calculus with impredicative polymorphism, iso-recursive types, higher-order state, and fine-grained concurrency. We start by describing the syntax (§2.1), typing (§2.2), and operational semantics (§2.3) of **MyLang**. Finally, we make the notion of type soundness formal (§2.4) and show how type soundness is proved using the standard syntactic approach (§2.5).

2.1 Syntax

We present the syntax of **MyLang** in two variations: *static* expressions $\hat{e} \in \widehat{Expr}$, and *dynamic* expressions $e \in Expr$. The idea behind this distinction is that the static syntax is used for writing surface programs, but in order for these programs to be executed, they must first be transformed by an *erasure* function $|\_| : \widehat{Expr} \to Expr$ into dynamic programs. Since ultimately we would like to prove the safety of programs that are not syntactically well-typed, throughout most of this paper we will work with dynamic syntax. In particular, we will define the operational semantics (§2.3) and our semantic type system (§4–§6) on the dynamic syntax. However, when presenting example programs, we use the static syntax, since these programs are written by users of **MyLang**.

The syntax of types, static expression, and dynamic expressions is shown in Figure 1. We let α range over *Tvar*, a countably infinite set of type variables, and let x and f range over *Var*, a countably infinite set of term variables.

The static expressions of **MyLang** are in Church style, *i.e.,* they include type annotations (marked in red). The dynamic, Curry-style syntax of **MyLang** is obtained by simply erasing all type annotations and by adding an additional literal $\ell \in Loc$ for memory locations. Locations only appear in the dynamic syntax because the programmer is not permitted to write them directly in the source program—they only get created dynamically during execution.

The types of **MyLang** include the ground types: the unit type $\mathbf{1}$, the type of Booleans $\mathbf{2}$, and the type of integers $\mathbf{Z}$. Basic type formers include products $(A \times B)$, sums $(A + B)$, and function types $(A \to B)$. Types also include iso-recursive types $(\mu\alpha.\, A)$, polymorphic types $(\forall\alpha.\, A)$, and existential types $(\exists\alpha.\, A)$, which classify *abstract data types* (or *ADTs*). The type $\texttt{ref}(A)$ is the type of memory locations that store values of type A.

Following Mitchell and Plotkin [1988], the expression $\texttt{pack}\,\langle B, \hat{e}\rangle : \exists\alpha.\, A$ represents an abstract data type (ADT), *i.e.,* an expression of existential type $(\exists\alpha.\, A)$, which "packs" the type "witness" B (representing the abstract type α) together with the term $\hat{e}$ (representing the operations on the

$$\begin{array}{rcll}
A, B \in \mathit{Type} & ::= & \alpha \in \mathit{Tvar} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{Z} \mid A \times A \mid A + A \mid A \to A \mid \forall \alpha.\, A \mid \exists \alpha.\, A \mid \mu\alpha.\, A \mid \texttt{ref}(A) & \\
\circledcirc \in \mathit{BinOp} & ::= & + \mid * \mid - \mid < \mid = & \\
\hat{e} \in \widehat{\mathit{Expr}} & ::= & x \in \mathit{Var} \mid \texttt{rec}\; f(x) = \hat{e} \mid \hat{e}\;\hat{e} \mid \Lambda\alpha.\,\hat{e} \mid \hat{e}\langle A\rangle \mid & \text{(Polymorphic } \lambda\text{-calculus)} \\
 & & () \mid n \in \mathbb{Z} \mid \hat{e} \circledcirc \hat{e} \mid & \text{(Unit type and arithmetic)} \\
 & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\; \hat{e}\; \texttt{then}\; \hat{e}\; \texttt{else}\; \hat{e} \mid & \text{(Booleans)} \\
 & & (\hat{e}, \hat{e}) \mid \pi_1\, \hat{e} \mid \pi_2\, \hat{e} \mid & \text{(Products)} \\
 & & \texttt{inj}_1\, \hat{e} \mid \texttt{inj}_2\, \hat{e} \mid (\texttt{match}\; \hat{e}\; \texttt{with}\; \texttt{inj}_1\, x \Rightarrow \hat{e}_1 \mid \texttt{inj}_2\, x \Rightarrow \hat{e}_2\; \texttt{end}) \mid & \text{(Sums)} \\
 & & \texttt{pack}\, \langle B, \hat{e}\rangle : \exists\alpha.\, A \mid \texttt{match}\; \hat{e}\; \texttt{with}\; \texttt{pack}\, \langle \alpha, x\rangle \Rightarrow \hat{e}\; \texttt{end} \mid & \text{(Existentials)} \\
 & & \texttt{fold}\; \hat{e} \mid \texttt{unfold}\; \hat{e} \mid & \text{(Iso-recursive types)} \\
 & & \texttt{ref}(\hat{e}) \mid !\, \hat{e} \mid \hat{e} \leftarrow \hat{e} \mid \texttt{CAS}(\hat{e}, \hat{e}, \hat{e}) \mid \texttt{FAA}(\hat{e}, \hat{e}) \mid & \text{(References)} \\
 & & \texttt{fork}\; \{\hat{e}\} & \text{(Concurrency)} \\
e \in \mathit{Expr} & ::= & x \in \mathit{Var} \mid \texttt{rec}\; f(x) = e \mid e\; e \mid \Lambda.\, e \mid e\langle\rangle \mid & \text{(Polymorphic } \lambda\text{-calculus)} \\
 & & () \mid n \in \mathbb{Z} \mid e \circledcirc e \mid & \text{(Unit type and arithmetic)} \\
 & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\; e\; \texttt{then}\; e\; \texttt{else}\; e \mid & \text{(Booleans)} \\
 & & (e, e) \mid \pi_1\, e \mid \pi_2\, e \mid & \text{(Products)} \\
 & & \texttt{inj}_1\, e \mid \texttt{inj}_2\, e \mid \texttt{match}\; e\; \texttt{with}\; \texttt{inj}_i\, x \Rightarrow e_i\; \texttt{end} \mid & \text{(Sums)} \\
 & & \texttt{pack}\langle e\rangle \mid \texttt{match}\; e\; \texttt{with}\; \texttt{pack}\langle x\rangle \Rightarrow e\; \texttt{end} \mid & \text{(Existentials)} \\
 & & \texttt{fold}\; e \mid \texttt{unfold}\; e \mid & \text{(Iso-recursive types)} \\
 & & \ell \in \mathit{Loc} \mid \texttt{ref}(e) \mid !\, e \mid e \leftarrow e \mid \texttt{CAS}(e, e, e) \mid \texttt{FAA}(e, e) \mid & \text{(References)} \\
 & & \texttt{fork}\; \{e\} & \text{(Concurrency)}
\end{array}$$

Fig. 1. Syntax of **MyLang**: Types A, B, binary operators $\circledcirc$, static expressions $\hat{e}$, and dynamic expressions e.

ADT of type A). The type witness is "abstract" in the sense that there is no way for clients of the ADT to observe the implementation of α as B. ADTs can be unpacked using a syntax similar to ML-style matches: $\texttt{match}\; \hat{e}_1\; \texttt{with}\; \texttt{pack}\, \langle \alpha, x\rangle \Rightarrow \hat{e}_2\; \texttt{end}$. Here, the term component of the ADT $\hat{e}_1$ can be referred to as x (and its type witness as α) within the scope of the expression $\hat{e}_2$.

The type operator $(\mu\alpha.\, A)$ for iso-recursive types describes a type that is isomorphic to $A[\mu\alpha.\, A/\alpha]$. For example, linked lists with elements of type B are given by $\texttt{linkedlist}(B) \triangleq \mu\alpha.\, \texttt{ref}(\mathbf{1}+(B\times\alpha))$, where the sum $+$ indicates that the list is either "nil" (*i.e.*, $\mathbf{1}$) or a "cons" (*i.e.*, $B\times\alpha$). The isomorphisms between $\mu\alpha.\, A$ and $A[\mu\alpha.\, A/\alpha]$ are witnessed by the operations $\texttt{fold}$ and $\texttt{unfold}$ as follows:

$$\begin{tikzcd}
\mu\alpha.\, A \arrow[r, bend left, "\texttt{unfold}"] & A[\mu\alpha.\, A/\alpha] \arrow[l, bend left, "\texttt{fold}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{linkedlist}(B) \arrow[r, bend left, "\texttt{unfold}"] & \texttt{ref}(\mathbf{1} + (B \times \texttt{linkedlist}(B))) \arrow[l, bend left, "\texttt{fold}"]
\end{tikzcd}$$

References can be allocated, read from, and written to using the $\texttt{ref}(\hat{e})$, $!\,\hat{e}$, and $\hat{e}_1 \leftarrow \hat{e}_2$ expressions, respectively. The compare-and-set $\texttt{CAS}$ and the fetch-and-add $\texttt{FAA}$ operations are **MyLang** primitives for fine-grained concurrency. The expression $\texttt{CAS}(\hat{e}_1, \hat{e}_2, \hat{e}_3)$ evaluates the three subexpressions to values v_1, v_2, and v_3, where v_1 must be a memory location ℓ; it then *atomically* checks if the value stored in memory at ℓ is equal to v_2, and, if so, updates ℓ to store v_3 instead. The expression

FAA$(\hat{e}_1, \hat{e}_2)$ *atomically* increments the value stored in the location described by $\hat{e}_1$ by the result of $\hat{e}_2$. The expression fork $\{\hat{e}\}$ forks a new thread to execute $\hat{e}$, while the current thread returns $()$.

Syntactic sugar. Non-recursive functions $\lambda\, x.\, e$ are defined as rec $\_(x) = e$, let-bindings let $x = e_1$ in e_2 are defined as $(\lambda\, x.\, e_2)\ e_1$, and sequential composition $e_1; e_2$ is defined as let $\_ = e_1$ in e_2. Here, we use the underscore _ to denote an anonymous binder, which is not used in the body of the binding expression.

2.2 Typing

We write $\Delta; \Gamma \vdash \hat{e} : A$ for the syntactic typing judgment, which expresses that an expression $\hat{e}$ of has type A under the typing contexts Γ and Δ. The typing context Γ is a list of the form $x_1 : A_1, \cdots, x_n : A_n$, which associates free variables (that may appear in $\hat{e}$) to their types. The type-level context Δ is a list $\alpha_1, \cdots, \alpha_m$ of free type variables that may appear in A or Γ. The empty variable and type-level context are denoted by $\emptyset$.

The syntactic typing rules of **MyLang** are displayed in Figure 2. The typing rule T-CAS for the CAS operation has the side-condition $\mathrm{EqType}(A)$, which ensures that a CAS can only be performed on word-sized data types. The rules for the EqType predicate are also displayed in Figure 2. The typing rule T-BINOP for binary operators $\circledcirc$ has the side-condition $\circledcirc : A_1 \times A_2 \Rightarrow B$, which expresses that when supplied with arguments of type A_1 and A_2, the operator has a result of type B. The rules are $\circledcirc : \mathbf{Z} \times \mathbf{Z} \Rightarrow \mathbf{Z}$ for $\circledcirc \in \{+, *, -\}$, and $(<) : \mathbf{Z} \times \mathbf{Z} \Rightarrow \mathbf{2}$, and $(=) : A \times A \Rightarrow \mathbf{2}$ for $\mathrm{EqType}(A)$.

We also define a typing judgment $\Delta; \Gamma \vdash e : A$ on dynamic expressions analogously to the typing judgment for static expressions. We omit the definition here for brevity; it can be derived from the typing judgment for static expressions by simply removing all the red text from Figure 2 and replacing all the $\hat{e}$'s with e's. It is then straightforward to show that $\Delta; \Gamma \vdash e : A$ if and only if there exists a static expression $\hat{e}$ such that $|\, \hat{e}\, | = e$ and $\Delta; \Gamma \vdash \hat{e} : A$.

2.3 Operational Semantics

To define the operational semantics of **MyLang**, we first define *values*, *states*, and *evaluation contexts*, as shown in Figure 3. These definitions are mostly standard. The states $\sigma \in \mathit{State}$ of **MyLang** are heaps, which we model as partial functions with finite support from memory locations to values. The evaluation contexts $K \in \mathit{Ctx}$ are used to define a left-to-right call-by-value (CBV) evaluation strategy for **MyLang**.

With these notions in hand, we define the small-step operational semantics of **MyLang** in three stages:

(1) We first define a *basic reduction* relation, $(\sigma, e) \rightarrow_{\mathrm{b}} (\sigma', e')$, which describes how e reduces under initial state σ to a new e' and (possibly) updated state σ'. This definition of the basic reduction relation makes use of the auxiliary *pure reduction* relation $e \rightarrow_{\mathrm{pure}} e'$ to handle state-independent reductions. The rules are shown in Figure 3. The function $[\![\circledcirc]\!] : \mathit{Val} \times \mathit{Val} \rightharpoonup \mathit{Val}$ assigns a denotation to each binary operator $\circledcirc$. This function is partial to account for operators that are applied wrongly, *e.g.,* $10\ [\![+]\!]$ true is undefined. We write $\uplus$ for the disjoint union operation on heaps.

(2) Following Felleisen and Hieb [1992], we use evaluation contexts to lift the basic reduction relation to a *thread-local reduction* relation $(\sigma, e) \rightarrow_{\mathrm{t}} (\sigma', e')$.

(3) Finally, the *thread-pool reduction* relation $(\sigma, \vec{e}) \rightarrow_{\mathrm{tp}} (\sigma', \vec{e'})$ for our programs is a relation defined on machine states, *i.e.,* a pair of a state and a thread pool (represented as a sequence of expressions $\vec{e}$ executing in different threads). The thread-pool reduction relation expresses that a machine state reduces by picking an arbitrary thread and either making a thread-local reduction step in that expression or else executing a fork and spawning a new thread.

T-Var
$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}$$

T-Unit
$$\Delta; \Gamma \vdash () : \mathbf{1}$$

T-Bool
$$\frac{b \in \{\mathtt{true}, \mathtt{false}\}}{\Delta; \Gamma \vdash b : \mathbf{2}}$$

T-Int
$$\frac{n \in \mathbb{Z}}{\Delta; \Gamma \vdash n : \mathbf{Z}}$$

T-Binop
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : A_1 \qquad \Delta; \Gamma \vdash \hat{e}_2 : A_2 \qquad \circledcirc : A_1 \times A_2 \Rightarrow B}{\Delta; \Gamma \vdash \hat{e}_1 \circledcirc \hat{e}_2 : B}$$

T-Rec
$$\frac{\Delta; \Gamma, x : A, f : A \to B \vdash \hat{e} : B}{\Delta; \Gamma \vdash \mathtt{rec}\, f(x) = \hat{e} : A \to B}$$

T-App
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : A \to B \qquad \Delta; \Gamma \vdash \hat{e}_2 : A}{\Delta; \Gamma \vdash \hat{e}_1\ \hat{e}_2 : B}$$

T-TLam
$$\frac{\Delta, \alpha; \Gamma \vdash \hat{e} : A}{\Delta; \Gamma \vdash \Lambda\alpha.\, \hat{e} : \forall\alpha.\, A}$$

T-TApp
$$\frac{\Delta; \Gamma \vdash \hat{e} : \forall\alpha.\, A \qquad \mathrm{FV}(B) \subseteq \Delta}{\Delta; \Gamma \vdash \hat{e}\langle B \rangle : A[B/\alpha]}$$

T-If
$$\frac{\Delta; \Gamma \vdash \hat{e} : \mathbf{2} \qquad \Delta; \Gamma \vdash \hat{e}_1 : B \qquad \Delta; \Gamma \vdash \hat{e}_2 : B}{\Delta; \Gamma \vdash \mathtt{if}\ \hat{e}\ \mathtt{then}\ \hat{e}_1\ \mathtt{else}\ \hat{e}_2 : B}$$

T-Pair
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : A_1 \qquad \Delta; \Gamma \vdash \hat{e}_2 : A_2}{\Delta; \Gamma \vdash (\hat{e}_1, \hat{e}_2) : A_1 \times A_2}$$

T-Proj
$$\frac{\Delta; \Gamma \vdash \hat{e} : A_1 \times A_2 \qquad i \in \{1, 2\}}{\Delta; \Gamma \vdash \pi_i\ \hat{e} : A_i}$$

T-Inj
$$\frac{\Delta; \Gamma \vdash \hat{e} : A_i \qquad i \in \{1, 2\}}{\Delta; \Gamma \vdash \mathtt{inj}_i\ \hat{e} : A_1 + A_2}$$

T-Match-Sum
$$\frac{\Delta; \Gamma \vdash \hat{e} : A_1 + A_2 \qquad \forall i \in \{1, 2\}\,.\ \Delta; \Gamma, x : A_i \vdash \hat{e}_i : B}{\Delta; \Gamma \vdash \mathtt{match}\ \hat{e}\ \mathtt{with}\ \mathtt{inj}_1\, x \Rightarrow \hat{e}_1 \mid \mathtt{inj}_2\, x \Rightarrow \hat{e}_2\ \mathtt{end} : B}$$

T-Pack
$$\frac{\Delta; \Gamma \vdash \hat{e} : A[B/\alpha] \qquad \mathrm{FV}(B) \subseteq \Delta}{\Delta; \Gamma \vdash \mathtt{pack}\ \langle B, \hat{e} \rangle : \exists\alpha.\, A : \exists\alpha.\, A}$$

T-Match-Ex
$$\frac{\Delta; \Gamma \vdash \hat{e} : \exists\alpha.\, A \qquad \Delta, \alpha; \Gamma, x : A \vdash \hat{e}_2 : B}{\Delta; \Gamma \vdash \mathtt{match}\ \hat{e}\ \mathtt{with}\ \mathtt{pack}\ \langle \alpha, x \rangle \Rightarrow \hat{e}_2\ \mathtt{end} : B}$$

T-Fold
$$\frac{\Delta; \Gamma \vdash \hat{e} : A[\mu\alpha.\, A/\alpha]}{\Delta; \Gamma \vdash \mathtt{fold}\ \hat{e} : \mu\alpha.\, A}$$

T-Unfold
$$\frac{\Delta; \Gamma \vdash \hat{e} : \mu\alpha.\, A}{\Delta; \Gamma \vdash \mathtt{unfold}\ \hat{e} : A[\mu\alpha.\, A/\alpha]}$$

T-Alloc
$$\frac{\Delta; \Gamma \vdash \hat{e} : A}{\Delta; \Gamma \vdash \mathtt{ref}(\hat{e}) : \mathtt{ref}(A)}$$

T-Load
$$\frac{\Delta; \Gamma \vdash \hat{e} : \mathtt{ref}(A)}{\Delta; \Gamma \vdash\ !\, \hat{e} : A}$$

T-Store
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : \mathtt{ref}(A) \qquad \Delta; \Gamma \vdash \hat{e}_2 : A}{\Delta; \Gamma \vdash \hat{e}_1 \leftarrow \hat{e}_2 : \mathbf{1}}$$

T-CAS
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : \mathtt{ref}(A) \qquad \Delta; \Gamma \vdash \hat{e}_2 : A \qquad \Delta; \Gamma \vdash \hat{e}_3 : A \qquad \mathsf{EqType}(A)}{\Delta; \Gamma \vdash \mathtt{CAS}(\hat{e}_1, \hat{e}_2, \hat{e}_3) : \mathbf{2}}$$

T-FAA
$$\frac{\Delta; \Gamma \vdash \hat{e}_1 : \mathtt{ref}(\mathbf{Z}) \qquad \Delta; \Gamma \vdash \hat{e}_2 : \mathbf{Z}}{\Delta; \Gamma \vdash \mathtt{FAA}(\hat{e}_1, \hat{e}_2) : \mathbf{Z}}$$

T-Fork
$$\frac{\Delta; \Gamma \vdash \hat{e} : A}{\Delta; \Gamma \vdash \mathtt{fork}\ \{\hat{e}\} : \mathbf{1}}$$

EqTyp-Unit
$$\mathsf{EqType}(\mathbf{1})$$

EqTyp-Bool
$$\mathsf{EqType}(\mathbf{2})$$

EqTyp-Int
$$\mathsf{EqType}(\mathbf{Z})$$

EqTyp-Ref
$$\mathsf{EqType}(\mathtt{ref}(A))$$

Fig. 2. Typing rules of **MyLang**.

Values, states, and evaluation contexts:

$$
\begin{aligned}
v \in \mathit{Val} ::={}& \mathtt{rec}\, f(x) = e \mid \Lambda.\, e \mid () \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \\
& (v, v) \mid \mathtt{inj}_1\, v \mid \mathtt{inj}_2\, v \mid \mathtt{pack}\langle v \rangle \mid \mathtt{fold}\, v \mid \ell \in \mathit{Loc}
\end{aligned}
$$

$$\sigma \in \mathit{State} \triangleq \mathit{Loc} \rightharpoonup_{\mathrm{fin}} \mathit{Val}$$

$$
\begin{aligned}
K \in \mathit{Ctx} ::={}& [\,] \mid K\ e \mid v\ K \mid K\langle\rangle \mid K \circledcirc e \mid v \circledcirc K \mid \mathtt{if}\ K\ \mathtt{then}\ e\ \mathtt{else}\ e \\
& (K, e) \mid (v, K) \mid \pi_1\, K \mid \pi_2\, K \mid \\
& \mathtt{inj}_1\, K \mid \mathtt{inj}_2\, K \mid (\mathtt{match}\ K\ \mathtt{with}\ \mathtt{inj}_1\, x \Rightarrow e_1 \mid \mathtt{inj}_2\, x \Rightarrow e_2\ \mathtt{end}) \mid \\
& \mathtt{pack}\langle K \rangle \mid \mathtt{match}\ K\ \mathtt{with}\ \mathtt{pack}\langle x \rangle \Rightarrow e\ \mathtt{end} \mid \\
& \mathtt{fold}\, K \mid \mathtt{unfold}\, K \mid \\
& \mathtt{ref}(K) \mid !\, K \mid K \leftarrow e \mid v \leftarrow K \mid \\
& \mathtt{CAS}(K, e, e) \mid \mathtt{CAS}(v, K, e) \mid \mathtt{CAS}(v, v, K) \mid \\
& \mathtt{FAA}(K, e) \mid \mathtt{FAA}(v, K)
\end{aligned}
$$

Pure reduction:

$$
\begin{array}{rcll}
(\mathtt{rec}\, f(x) = e)\ v & \to_{\mathrm{pure}} & e[v/x][\mathtt{rec}\, f(x) = e/f] & \\
(\Lambda.\, e)\langle\rangle & \to_{\mathrm{pure}} & e & \\
v_1 \circledcirc v_2 & \to_{\mathrm{pure}} & v_1\ [\![\circledcirc]\!]\ v_2 & \\
\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 & \to_{\mathrm{pure}} & e_1 & \\
\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 & \to_{\mathrm{pure}} & e_2 & \\
\pi_i\ (v_1, v_2) & \to_{\mathrm{pure}} & v_i & (\text{if } i \in \{1, 2\}) \\
\mathtt{match}\ \mathtt{inj}_i\, v\ \mathtt{with}\ \mathtt{inj}_1\, x \Rightarrow e_1 \mid \mathtt{inj}_2\, x \Rightarrow e_2\ \mathtt{end} & \to_{\mathrm{pure}} & e_i[v/x] & (\text{if } i \in \{1, 2\}) \\
\mathtt{match}\ \mathtt{pack}\langle v \rangle\ \mathtt{with}\ \mathtt{pack}\langle x \rangle \Rightarrow e\ \mathtt{end} & \to_{\mathrm{pure}} & e[v/x] & \\
\mathtt{unfold}\ (\mathtt{fold}\, v) & \to_{\mathrm{pure}} & v &
\end{array}
$$

Basic reduction:

$$
\begin{array}{rcll}
(\sigma, e_1) & \to_{\mathrm{b}} & (\sigma, e_2) & (\text{if } e_1 \to_{\mathrm{pure}} e_2) \\
(\sigma, \mathtt{ref}(v)) & \to_{\mathrm{b}} & (\sigma \uplus \{(\ell, v)\}, \ell) & (\text{if } \ell \notin \mathrm{dom}(\sigma)) \\
(\sigma, !\, \ell) & \to_{\mathrm{b}} & (\sigma, v) & (\text{if } (\ell, v) \in \sigma) \\
(\sigma \uplus \{(\ell, v)\}, \ell \leftarrow w) & \to_{\mathrm{b}} & (\sigma \uplus \{(\ell, w)\}, ()) & \\
(\sigma \uplus \{(\ell, v)\}, \mathtt{CAS}(\ell, v, u)) & \to_{\mathrm{b}} & (\sigma \uplus \{(\ell, u)\}, \mathtt{true}) & \\
(\sigma \uplus \{(\ell, v)\}, \mathtt{CAS}(\ell, w, u)) & \to_{\mathrm{b}} & (\sigma \uplus \{(\ell, v)\}, \mathtt{false}) & (\text{if } v \neq w) \\
(\sigma \uplus \{(\ell, n)\}, \mathtt{FAA}(\ell, m)) & \to_{\mathrm{b}} & (\sigma \uplus \{(\ell, n + m)\}, n) &
\end{array}
$$

Thread-local and thread-pool reduction:

$$
\frac{(\sigma, e) \to_{\mathrm{b}} (\sigma', e')}{(\sigma, K[e]) \to_{\mathrm{t}} (\sigma', K[e'])}
\qquad\qquad
\frac{(\sigma, e) \to_{\mathrm{t}} (\sigma', e')}{(\sigma, (\vec{e_1}; e; \vec{e_2})) \to_{\mathrm{tp}} (\sigma', (\vec{e_1}; e'; \vec{e_2}))}
$$

$$(\sigma, (\vec{e_1}; K[\mathtt{fork}\ \{e\}\,]; \vec{e_2})) \to_{\mathrm{tp}} (\sigma, (\vec{e_1}; K[()]; \vec{e_2}; e))$$

Fig. 3. Operational semantics of **MyLang**.

2.4 Type Soundness

A programming language is *type-sound* (or *type-safe*) if every closed well-typed expression is safe (*i.e.,* has well-defined behavior). To define this more formally, we first give some auxiliary definitions:

- We say that a machine state $(\sigma, \vec{e})$ is *progressive*, written progressive$(\sigma, \vec{e})$, if any thread in that state is either a value (*i.e.,* it has finished executing), or it is *reducible* (*i.e.,* it can make at least one further step of computation):

$$\mathrm{progressive}(\sigma, (e_1; \ldots; e_n)) \triangleq \forall i \in \{1, \ldots, n\}.\, (e_i \in \mathit{Val} \vee \mathrm{red}(\sigma, e_i))$$
$$\mathrm{red}(\sigma, e) \triangleq (\exists \sigma', e'.\, (\sigma, e) \to_{\mathrm{t}} (\sigma', e')) \vee (\exists K, e'.\, e = K[\mathtt{fork}\ \{e'\}\,])$$

- We then say that a closed expression e, representing a complete program, is *safe*, written safe(e), if any machine state reachable by evaluating e for any number of steps is progressive:

$$\mathrm{safe}(e) \triangleq \forall \sigma_2, \vec{e_2}.\, (\emptyset, e) \to^{*}_{\mathrm{tp}} (\sigma_2, \vec{e_2}) \Rightarrow \mathrm{progressive}(\sigma_2, \vec{e_2})$$

Now we can formally define type soundness as:

$$(\emptyset; \emptyset \vdash e : A) \quad \text{implies} \quad \mathrm{safe}(e)$$

2.5 Syntactic Type Soundness via Progress and Preservation

The syntactic approach to proving type soundness involves two key theorems:

1. **Progress (Theorem 2.1).** Well-typed machine states are progressive.
2. **Preservation (Theorem 2.2).** Reduction preserves well-typedness of machine states.

These theorems rely on a notion of a *well-typed machine state* $(\sigma, \vec{e})$, which intuitively expresses that each value in the heap σ is well-typed and each expression in the thread-pool $\vec{e}$ is well-typed. To formalize this notion, we need to account for location literals ℓ. While location literals do not appear in static expressions, they may appear in runtime expressions and values during reduction (when memory cells are allocated), and their types need to match up with the types of values in the heap σ. We thus define a generalized typing judgment, written $\Sigma; \Delta; \Gamma \vdash e : A$, which extends the typing judgment $\Delta; \Gamma \vdash e : A$ with a *heap typing* $\Sigma : \mathit{Loc} \rightharpoonup_{\mathrm{fin}} \mathit{Type}$. A heap typing is a partial function with finite support that assigns a closed type to each location. The essential rule of the generalized typing judgment is the one for location literals:

$$\textsc{DT-loc}\quad \frac{\Sigma(\ell) = A}{\Sigma; \Delta; \Gamma \vdash \ell : \mathtt{ref}(A)}$$

In all rules but the one for location literals above, the heap typing is simply threaded through. For example, the rules for function application and allocation become:

$$\textsc{DT-app}\quad \frac{\Sigma; \Delta; \Gamma \vdash e_1 : A \to B \qquad \Sigma; \Delta; \Gamma \vdash e_2 : A}{\Sigma; \Delta; \Gamma \vdash e_1\ e_2 : B} \qquad\qquad \textsc{DT-alloc}\quad \frac{\Sigma; \Delta; \Gamma \vdash \hat{e} : A}{\Sigma; \Delta; \Gamma \vdash \mathtt{ref}(\hat{e}) : \mathtt{ref}(A)}$$

We can now define the notion of well-typed machine states with the judgment $\Sigma \vdash_{\mathrm{MS}} (\sigma, \vec{e}) : \vec{A}$:

$$\frac{\begin{array}{cc} \mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma) & \mathrm{length}(\vec{e}) = \mathrm{length}(\vec{A}) \\ \forall \ell \in \mathrm{dom}(\Sigma).\, \Sigma; \emptyset; \emptyset \vdash \sigma(\ell) : \Sigma(\ell) & \forall i < \mathrm{length}(\vec{e}).\, \Sigma; \emptyset; \emptyset \vdash e_i : A_i \end{array}}{\Sigma \vdash_{\mathrm{MS}} (\sigma, \vec{e}) : \vec{A}}$$

This judgment says that each value $\sigma(\ell)$ in σ has the corresponding type $\Sigma(\ell)$ from the heap typing Σ, and that, under Σ, each expression e_i in $\vec{e}$ has the corresponding type A_i from the list $\vec{A}$.

Theorem 2.1 (Progress). *Every machine state* $(\sigma, \vec{e})$ *that is typed for some heap environment* Σ *is safe. Formally, if* $\Sigma \vdash_{\mathrm{MS}} (\sigma, \vec{e}) : \vec{A}$, *then* $\mathrm{progressive}(\sigma, \vec{e})$.

Theorem 2.2 (Preservation). *Typing of machine states is preserved by the reduction relation* $\rightarrow_{\mathrm{tp}}$. *Formally, if* $\Sigma \vdash_{\mathrm{MS}} (\sigma, \vec{e}) : \vec{A}$ *and* $(\sigma, \vec{e}) \rightarrow_{\mathrm{tp}} (\sigma', \vec{e'})$, *then there exists an extended heap typing* $\Sigma' \supseteq \Sigma$ *and an extended list of thread types* $\vec{A'} \sqsupseteq_{\mathrm{prefix}} \vec{A}$ *such that* $\Sigma' \vdash_{\mathrm{MS}} (\sigma', \vec{e'}) : \vec{A'}$.

Progress and preservation are typically proven by induction and/or straightforward case analysis on the given typing and reduction derivations. These proofs involve some easy helping lemmas related to substitution and weakening w.r.t. extension of the heap typing, as well as (in some variations) lemmas for decomposing a well-typed term into a well-typed evaluation context and a well-typed redex. The interested reader can find a detailed account of the proofs in the course lecture notes of Dreyer et al. [2022]. For this paper, what is important is that progress and preservation give rise to the following result:

Corollary 2.3 (Syntactic type soundness). *Every closed well-typed expression e is safe. Formally, if* $(\emptyset; \emptyset \vdash e : A)$, *then* $\mathrm{safe}(e)$.

Proof. Assume that we have $(\emptyset; \emptyset \vdash e : A)$, and that we are given a reduction $(\emptyset, e) \rightarrow_{\mathrm{tp}}^{*} (\sigma_2, \vec{e_2})$. Our goal is to prove $\mathrm{progressive}(\sigma_2, \vec{e_2})$.

By definition of the judgment for well-typed machine states, we obtain $\emptyset \vdash_{\mathrm{MS}} (\emptyset, e) : A$ from the assumption $(\emptyset; \emptyset \vdash e : A)$. By repeatedly using preservation (Theorem 2.2) for each reduction step in $(\emptyset, e) \rightarrow_{\mathrm{tp}}^{*} (\sigma_2, \vec{e_2})$, we obtain $\Sigma \vdash_{\mathrm{MS}} (\sigma_2, \vec{e_2}) : \vec{A'}$ for some Σ and $\vec{A'}$. By progress (Theorem 2.1), we obtain $\mathrm{progressive}(\sigma_2, \vec{e_2})$, which concludes the proof. □

3 LIMITATIONS OF SYNTACTIC TYPE SOUNDNESS

Now that we have reviewed a typical formulation of syntactic type soundness, we are in a position to expound our criticisms of it:

(1) It says nothing about whether a type system enforces *data abstraction* (§3.1).
(2) It says nothing about programs that use *unsafe features* in a *safely encapsulated* way (§3.2).

3.1 Data Abstraction

Consider the following type `symbol_type`, which describes an (extremely simplified) interface of an abstract data type (ADT) of *symbols*:

$$\texttt{symbol\_type} \triangleq \exists \alpha.\, (\mathbf{1} \rightarrow \alpha) \times (\alpha \rightarrow \mathbf{2})$$

Following Mitchell and Plotkin [1988], we model the type of an ADT using an *existential type*.[4] Here, the existential type describes the interface of an ADT that exports an abstract type α representing "symbols", along with two operations: a `gensym` function of type $\mathbf{1} \rightarrow \alpha$ with which one can generate fresh symbols, and a `check` function of type $\alpha \rightarrow \mathbf{2}$ which one can use to check if a symbol is valid. This interface is obviously not very useful in the stripped-down form presented here, but it gives the flavor of the interface one often sees for symbol tables in compilers and will suffice to get across our point about syntactic type soundness.

Now, consider the implementation `symbol` of the `symbol_type` interface:

$$\begin{aligned}\texttt{symbol} \triangleq\ & \texttt{let}\ c = \texttt{ref}(0)\ \texttt{in} \\ & \texttt{pack}\left\langle \mathbf{Z}, \begin{pmatrix} \lambda\,().\, \texttt{FAA}(c, 1), \\ \lambda\, s.\, s < !\,c \end{pmatrix} \right\rangle : \texttt{symbol\_type}\end{aligned}$$

[4] See Rossberg et al. [2014] for an explanation of how existential types provide a foundation for understanding and formally modeling much more complex data abstraction facilities, such as those of the ML module system [MacQueen 1984].

This implementation employs a private integer counter c, which is allocated when the expression defining symbol is evaluated. The counter c is used as a perpetual source of fresh symbols. When the gensym function (the first closure returned by symbol) is called, it uses fetch-and-add (FAA) to atomically increment the value of c and return the previous value. Thus, when called repeatedly, gensym will return 0, 1, 2, *etc.* The check function (the second closure returned by symbol) checks validity of its symbol argument by checking that it is less than the current value of the counter.

It is easy to argue by appeal to intuitive reasoning that the check function returned by symbol must always return true. To see this, we first observe that the only values of the abstract type symbol (*i.e.,* α) that can ever be generated are those returned by the gensym function. However, whenever such a value is returned by gensym, it is at that instant one less than the current value of the counter c. Furthermore, the value of the counter only increases over time. Put together, these imply that, at all times, all values of type symbol are always less than the current value of the counter, so check applied to a value of type symbol should always return true.

This is the kind of informal reasoning about program correctness that programmers employ all the time. Crucially, though, it relies on the assumption that the programming language properly enforces two forms of data abstraction: *private state* (via *closures*) and *abstract types*. To enforce the invariant that the value of the counter c only increases over time, it is essential that the only way to modify c is by applying one of the closures returned by symbol—*i.e.,* that c is maintained as private state of those closures. Otherwise, clients could update c willy-nilly, and thus break the invariant. To enforce the invariant that the only values of the symbol type are the ones produced by calls to the gensym function, it is essential that the representation of the symbol type as $\mathbb{Z}$ be held abstract from clients. Otherwise, clients could take an arbitrary integer and "forge" a value of type symbol from it, which could cause the check function to return false.

Fortunately, it turns out that the language **MyLang** *does* properly enforce data abstraction, so the above informal reasoning is in fact valid. Unfortunately, proper treatment of data abstraction is not in any way a consequence of syntactic type soundness. To demonstrate this point, we will extend **MyLang** with a new and rather devious feature that we call, for lack of a better name, gremlin. This new feature has the property that on the one hand it is "harmless" in that it preserves the syntactic type soundness of **MyLang**, but on the other hand it completely breaks the language's support for data abstraction and hence the ability to reason modularly about **MyLang** code.

The static and dynamic semantics of gremlin are as follows:

$$\Delta; \Gamma \vdash \texttt{gremlin} : \mathbf{1}$$

$$(\sigma, \texttt{gremlin}) \to_{\mathsf{b}} (\sigma, ()) \qquad\qquad (\sigma \uplus \{(\ell, n)\}, \texttt{gremlin}) \to_{\mathsf{b}} (\sigma \uplus \{(\ell, 0)\}, ())$$

In short, gremlin is an expression of type $\mathbf{1}$, and its execution non-deterministically proceeds in one of two ways. Either it is simply a no-op, or else it non-deterministically selects some memory location ℓ currently storing an integer value n, and it updates ℓ to store 0.

As far as syntactic type soundness is concerned, gremlin is almost trivially a "safe" operator. First, the no-op evaluation rule ensures that gremlin can always make progress. Second, if gremlin does have an effect, it is merely to replace the integer value stored at some memory location with another integer value (namely, 0), thus preserving syntactic well-typedness of the heap. Consequently, it is easy to extend the syntactic soundness result from §2.5 to account for gremlin.

However, as should be intuitively clear, gremlin is a *terrible* feature because it destroys the programmer's ability to place invariants on the private state of their ADTs. To make this point perfectly concrete, consider the following client of the symbol ADT:

```
evil_client ≜ match symbol with pack ⟨$\alpha$,$x$⟩ ⇒
                  let gensym = $\pi_1$ $x$ in
                  let check = $\pi_2$ $x$ in
                  let $s$ = gensym () in
                  gremlin; check $s$
              end
```

After unpacking the existential representing the ADT, the client first calls the gensym function to create a fresh symbol value s. It then invokes gremlin, and finally calls check on s.

When this client is executed, the call to gensym will have the effect of updating the ADT's private counter c to 1, and returning the value 0 for s. When gremlin is invoked, one possible behavior is that the counter c will be set back to 0. If that happens, the subsequent application of check to s (*i.e.,* to 0) will return false! The problem here, of course, is that with gremlin in the language, the private state of the symbol ADT is no longer truly private since gremlin can modify it. As a result, the programmer cannot depend on any invariants on the private counter being maintained.

Now gremlin may seem like a rather contrived operator, but it is actually just an absurdly pointless variation of what is already supported, for example, by the Reflection API in Java. Using reflection, one can inspect the private fields and methods of an arbitrary object and freely modify its private state [Nasi 2011], thus achieving the same devastating effect on data abstraction as gremlin does.

Fortunately, since version 9, Java has given programmers the choice of whether the private state of their ADTs should be accessible via reflection from other ADTs. (Prior to Java 9, there was no way to limit reflective access.) Consequently, we would really like to be able to prove *some* theorem about data abstraction in Java 9 that would not hold of prior versions of the language. But seeing as reflection—like gremlin—is perfectly "type-safe" in the syntactic sense, syntactic type soundness is not that theorem.

In summary, syntactic type soundness of a programming language tells us essentially nothing about whether the language is sensible to program in.

3.2 Safe Encapsulation of Unsafe Features

As noted in the introduction, the price that "safe" programming languages pay for safety is that their type systems do not always allow the programmer to write the code they want to write. Consequently, most such languages provide *unsafe escape hatches* by which programmers can circumvent the restrictions of their type systems. For example, OCaml provides Obj.magic, an unchecked type cast operator. Haskell provides unsafeCoerce (similar to Obj.magic), unsafePerformIO (for escaping the IO monad), and more. Rust provides a whole host of low-level C-style operations, although uses of them must be confined to blocks of code explicitly marked unsafe. These unsafe escape hatches are widely used and depended upon in real-world applications.

Of course, given that these unsafe features are blatantly dangerous, programmers are advised to only make use of them if "you know what you are doing", and a major aspect of "knowing what you are doing" is knowing how to use unsafe features in a *safely encapsulated* way. That is, when a programmer uses unsafe features in the implementation of some ADT M, they typically rely on the data abstraction mechanisms of the programming language to enforce invariants on the private state or data representation of M, invariants which imply that the local uses of unsafe features within M will never lead to any undefined behavior for M's clients. In this way, one can see type systems as their own saving grace: though their limitations sometimes necessitate the use of unsafe workarounds, it is their data abstraction mechanisms that make it possible to use those workarounds "locally"—*i.e.*, without breaking the safety guarantees of the language as a whole.

Hence, the ability to safely encapsulate uses of unsafe features is inextricably linked with data abstraction: understanding whether a programming language supports one is tantamount to understanding whether it supports the other. To drive this point home, let us explain how we can recast the example of the symbol ADT from §3.1 in terms of safe encapsulation of unsafe features.

First, let us extend (the static and dynamic) syntax of **MyLang** with a new feature: *assertions*, written $\mathtt{assert}\ e$. The dynamic semantics of assertion expressions is given as follows:

$$K ::= \ldots \mid \mathtt{assert}\ K \qquad\qquad \mathtt{assert\ true} \rightarrow_{\mathsf{pure}} \mathtt{true}$$

In words, $\mathtt{assert}\ e$ will first evaluate e to a value v, and then the entire assertion will evaluate to $\mathtt{true}$ iff $v = \mathtt{true}$. However, if v evaluates to anything else, $\mathtt{assert}\ e$ will get stuck.[5] Thus, $\mathtt{assert}$ is only safe to use if applied to an expression that is guaranteed to evaluate to $\mathtt{true}$. As such, $\mathtt{assert}$ is in general an unsafe feature and, indeed, it is *not* syntactically well-typed.

Now consider the following, slightly revised implementation of the symbol ADT from §3.1 (the changed code is underlined):

$$\mathtt{symbol} \triangleq \mathtt{let}\ c = \mathtt{ref}(0)\ \mathtt{in}$$
$$\mathtt{pack}\left\langle \mathbb{Z}, \begin{pmatrix} \lambda\,().\ \mathtt{FAA}(c, 1), \\ \lambda\,s.\ \underline{\mathtt{assert}\ (s < !\,c)} \end{pmatrix} \right\rangle : \mathtt{symbol\_type}$$

Rather than simply returning the result of $s < !\,c$, the check function now $\mathtt{assert}$s it. Consequently, check will now only be safe to execute (*i.e.*, not get stuck) if $s < !\,c$ indeed evaluates to $\mathtt{true}$. As we have already mentioned in §3.1, **MyLang**'s support for data abstraction *should* ensure that $s < !\,c$ *does* always evaluate to $\mathtt{true}$ in all well-typed contexts—or equivalently, that the new symbol ADT has safely encapsulated the potentially unsafe behavior of the $\mathtt{assert}$ expression in its body, so that no well-typed client of symbol will ever encounter undefined (stuck) behavior. But syntactic type soundness does not offer us a means to prove this! In the next section, we will see a more powerful approach to formalizing type soundness that does.

4 SEMANTIC TYPE SOUNDNESS

In this section, we explain how to overcome the limitations of syntactic type soundness (described in §3) via the alternative approach of *semantic type soundness*. We begin with a brief high-level explanation of how a semantic type soundness proof is structured (§4.1). We then discuss prior approaches to semantic type soundness and the problems they suffer, which might explain why such approaches have not been more widely adopted (§4.2 and §4.3). We conclude by explaining how Iris addresses these problems, thus providing an ideal logical framework in which semantic type soundness can be more effectively formalized (§4.4). The sections that follow (§5–§7) will then present our *logical approach* to proving semantic type soundness in great detail.

4.1 High-Level Overview of Semantic Type Soundness

The central problem with syntactic type soundness is that it identifies "safe" with "syntactically well-typed". As a result, it is unable to account for the safety of code that uses potentially unsafe features in a well-encapsulated way, such as the example at the end of §3.2.

To overcome this problem, semantic type soundness models safety instead using a more liberal view of well-typedness, which we call *semantic typing* and write as $\Delta; \Gamma \vDash e : A$. The difference is that, whereas syntactic typing is *intensional* (it dictates a fixed set of syntactic rules by which safe

[5] One can in fact encode a primitive extremely similar to $\mathtt{assert}\ e$ in **MyLang** without any extension, via the following syntactic sugar: $\mathtt{assert}\ e \triangleq \mathtt{if}\ e\ \mathtt{then\ true\ else}\ 42(42)$.

terms can be constructed), semantic typing is *extensional* (it merely requires that terms behave in a safe way when executed). For example, the `symbol` ADT from §3.2, though not syntactically well-typed due to its use of the unsafe `assert` expression, will be shown to be semantically well-typed at the type `symbol_type`, thus establishing that `symbol` is in fact safe to use at that type. Of course, the price paid for this extensionality is that semantic typing is in general not a property that can be checked algorithmically. Rather, proving that a term is semantically well-typed may require arbitrarily interesting verification effort. But this is to be expected given that the goal of semantic soundness is to help establish that ADTs are properly maintaining their internal invariants, a task which often amounts to proving full functional correctness of the code.

The high-level structure of a semantic type soundness proof is simple:

- **Adequacy.** First, we prove an *adequacy* theorem, which establishes that closed semantically well-typed terms are indeed safe to execute. Formally, this means that $\emptyset; \emptyset \vDash e : A$ implies $\mathrm{safe}(e)$. This theorem is usually almost trivial to prove because, as explained above, it is typically more or less baked into the extensional definition of semantic typing.
- **Semantic typing rules.** Second, and more interestingly, we prove semantic versions of all the syntactic typing rules of the language, where the semantic version of a typing rule simply replaces all the syntactic $\vdash$'s in it with semantic $\vDash$'s. For instance, concerning function applications in the language **MyLang**, we will prove the following *semantic typing rule* as a lemma stating that the premises imply the conclusion:

$$\frac{\Delta; \Gamma \vDash e_1 : A \to B \qquad \Delta; \Gamma \vDash e_2 : A}{\Delta; \Gamma \vDash e_1\ e_2 : B}$$

  These semantic typing rules serve to demonstrate that semantic typing is compositional in the same way that syntactic typing is.

One immediate consequence of the semantic typing rules is that syntactic typing implies semantic typing, *i.e.,* $\Delta; \Gamma \vdash e : A$ implies $\Delta; \Gamma \vDash e : A$. Historically, this property is often referred to as the *fundamental theorem* or *fundamental property*. (It is provable by a straightforward induction on syntactic typing derivations.) As a result, closed syntactically well-typed programs are also semantically well-typed, and by adequacy, they must be safe to execute. In other words, *once we have proven semantic type soundness, syntactic type soundness falls out as a simple corollary.*

But the main reason we care about semantic type soundness is that it is a strictly *stronger* result than syntactic type soundness: it shows that we can safely compose syntactically *well*-typed pieces of a program with other pieces that are syntactically *ill*-typed (*e.g.,* use unsafe features) so long as those other pieces are *semantically well-typed*. For instance, once we prove that the `symbol` ADT is semantically well-typed at the type `symbol_type`, we will be able to deduce that if `symbol` is used within any syntactically (or semantically) well-typed program context C, then the resulting whole program C[`symbol`] will be safe to execute—implying that the assertion inside `symbol`'s `check` function will always succeed.

4.2 Prior Work on Semantic Type Soundness

As noted in the introduction, there is a great deal of prior work on semantic type soundness, dating back to Milner [1978]'s original paper in which the idea of type soundness was introduced. However, the approach has never really "taken off" as a method for proving type soundness of more realistic languages in the same way that syntactic type soundness has. We now explain why we believe this is so, as it helps to provide a clearer motivation for the "logical" formulation of type soundness that is our main contribution.[6]

Milner [1978]'s original semantic soundness proof for a core ML-like calculus, as well as subsequent semantic soundness proofs for more expressive type systems with higher-order polymorphism, recursive types, and subtyping—*e.g.,* [MacQueen et al. 1986; Bruce and Mitchell 1992]—were formulated using "realizability" models, in which types were interpreted as certain kinds of subsets or partial equivalence relations over a domain-theoretic (*i.e.,* denotational) model of untyped computation. Such models were also used to study parametricity and data abstraction, both for functional languages with polymorphic types—*e.g.,* [Bainbridge et al. 1990; Abadi and Plotkin 1990]—and for imperative languages with local variables—*e.g.,* [O'Hearn and Tennent 1992]. However, developing denotational semantics for programming languages with higher-order state (*i.e.,* general mutable references to values of arbitrary type) turned out to be quite challenging. Indeed, despite the ubiquity of higher-order state in programming languages for the past several decades, it was only in the work of Birkedal et al. [2010] that the realizability approach over domains was finally extended to handle this feature.

In much of this line of work, it was understood implicitly that, due to the inherent compositionality of denotational models, they could serve to establish the safety of combining syntactically well-typed programs with syntactically unsafe, but semantically well-typed, programs (as we described in §4.1). But this capability was not commonly exploited or even remarked upon, perhaps because the focus was on building semantic models of higher-order richly-typed λ-calculi, not on modeling realistic languages with low-level unsafe primitives (such as the language studied in the RustBelt project [Jung et al. 2018a, 2021; Jung 2020; Dang et al. 2020]).

Unfortunately, this state of affairs has meant that, for realistic languages, Milner-style semantic soundness based on denotational semantics has not really offered a viable solution to the problem we posed in the introduction. And for the more modest goal of simply proving well-defined behavior for syntactically well-typed programs, progress and preservation has offered a more elementary and broadly applicable technique.

At the same time, there arose a related but distinct line of work on building semantic models of typed languages over an *operational* semantics rather than a denotational one. In particular, partial equivalence relations over operational semantics were used early on in seminal work on the NuPRL type theory [Constable et al. 1986; Allen 1987]. This approach was further developed to account for recursive types [Birkedal and Harper 1999], local state [Pitts and Stark 1998], and the combination of recursive types and polymorphism [Crary and Harper 2007]. The approach to recursive types in [Birkedal and Harper 1999; Crary and Harper 2007] employed a syntactic adaptation of the denotational idea of *minimal invariance* [Pitts 1996], but this was quite technically involved and, as with denotational methods, it was for a long time not clear how to generalize the approach to handle higher-order state.

An important breakthrough came in 2001 when Appel and McAllester [2001] developed their *step-indexed* model of recursive types. The basic idea of step-indexing is to stratify the quasi-circular definition of semantic typing for recursive types by the number of steps for which the term in question is allowed to execute.[7] One immediate benefit of step-indexing was that it supplied a much more elementary model of recursive types than the previous approaches based on minimal invariance. But the more important benefit of step-indexing was that the basic idea scaled to account for more complex features that were beyond the scope of denotational models. In particular, Ahmed

[6] The literature on semantics of type systems is voluminous, so this section should not be viewed as a comprehensive survey, but rather a brief dive into the literature in order to suggest where existing approaches to semantic type soundness come up short. See §9 for additional discussion.

[7] See Ahmed [2004] for a more detailed exposition of step-indexing.

in her PhD thesis [Ahmed 2004] (building on prior work with Appel and Virga [Ahmed et al. 2002]) showed how to apply step-indexing in a more sophisticated fashion in order to construct a semantic model of higher-order state.

Ahmed's thesis led in turn to a flood of follow-on work, including [Appel et al. 2007; Ahmed et al. 2009; Neis et al. 2009; Benton and Hur 2009; Dreyer et al. 2010; Birkedal et al. 2011; Krishnaswami and Benton 2011; Schwinghammer et al. 2013; Dreyer et al. 2012; Thamsborg and Birkedal 2011; Birkedal et al. 2012; Turon et al. 2013b; Birkedal et al. 2013]. This line of work has demonstrated that step-indexing—in conjunction with various other techniques, notably *biorthogonality* [Krivine 1994; Pitts and Stark 1998] and *Kripke logical relations* [Jung and Tiuryn 1993]—could be used to construct operational-semantics-based models of much more realistic languages, featuring (among other things) control effects, substructural types, intensional polymorphism, and concurrency. What is more, some of these models (*e.g.,* [Ahmed et al. 2009; Dreyer et al. 2010, 2012; Schwinghammer et al. 2013])[8] were developed for the express purpose of verifying the kind of invariants on the private state of ADTs that we saw in the `symbol` example from §3.1. (In fact, that example is adapted from one proven by Ahmed et al. [2009].) Other models used step-indexing and semantic soundness to reason about low-level code, *e.g.,* to capture what it means for a piece of low-level code to implement a high-level function and to prove correctness of a simple compiler [Benton and Hur 2009; Hur and Dreyer 2011].

These more sophisticated semantic models—constructed using step-indexing and companion techniques—are often referred to as *step-indexed Kripke logical-relations* (or *SKLR*) models. At this point, the reader may rightly wonder: if SKLR models already address the limitations of syntactic type soundness from §3, why are we writing this article? And if they are so powerful, why have they not gained widespread adoption? Why does syntactic type soundness continue to be much more commonly known and used?

4.3 The Problems with SKLR Models

Based on our personal experience with these SKLR models, we believe the reason they have not been more widely adopted is that if one works *directly* with these models, one's proofs become painfully tedious, low-level, and difficult to maintain. Specifically:

1. **Explicit step-indexed arithmetic:** When working directly in a step-indexed model of any kind, one ends up performing a great deal of tedious "step-index arithmetic"—*i.e.,* counting of how many computation steps different operations take—even though it seems for the most part completely irrelevant to what one is proving.[9]
2. **Explicit reasoning about global state:** When working directly in an SKLR model for a stateful language, one ends up reasoning explicitly about the global state of memory, even though the operations one is reasoning about only affect local pieces of that memory (*e.g.,* a single location).[10]
3. **Explicit reasoning about possible worlds:** When working directly in one of the more advanced SKLR models, one ends up performing a lot of tedious manipulation of and quantification over "possible worlds", which describe the set of invariants that have been established on the program state.[11]

[8] The cited works formalized invariants on private state *relationally—i.e.,* by proving certain kinds of *contextual refinements*—rather than in the setting of semantic soundness. We find it useful to start first in this section by formalizing such invariants in the simpler "unary" setting of semantic soundness, before showing in §8 how our approach generalizes to the more complex "binary" relational setting of prior work.

[9] See for instance the proofs in [Ahmed 2004] or [Ahmed et al. 2009].

[10] See for instance the proofs in [Ahmed et al. 2009] or [Schwinghammer et al. 2013].

[11] See for instance the proofs in [Ahmed et al. 2009], [Schwinghammer et al. 2013], or [Turon et al. 2013b].

For a representative example of this, we refer the reader to Ahmed's PhD thesis [Ahmed 2004] and the technical appendices accompanying several of her papers, *e.g.*, [Ahmed 2006]. Her formal developments are unusual (and commendable) in that they spell out step-indexing-based proofs in full, and with great attention to detail.[12] The end result, however, is that her proofs are cluttered with seemingly unnecessary low-level technical details about step-indexing, the global state, and possible worlds.[13] For example, see Ahmed's proof of the rule mentioned earlier in this section—the semantic typing rule for function applications—which appears as Theorem 3.21 in her thesis. The proof involves explicit step-index arithmetic throughout (*e.g.*, "let $k^* = k - j - i - 1$"), as well as manipulation of three global states and four possible worlds—and that is all for a semantic typing rule that has nothing to do with mutable state! As a result, compared with the cases of a progress-and-preservation proof concerning function applications, the semantic soundness proof appears significantly more low-level and complex, and for no clear reason.

This problem was noted fairly early on in the development of SKLR models [Appel et al. 2007], and led to a fruitful line of work on *program logics* for encoding SKLR models at a much higher level of abstraction [Dreyer et al. 2011, 2010; Turon et al. 2013a]. In combination with a line of work on higher-order concurrent separation logic [Svendsen et al. 2013; Svendsen and Birkedal 2014], this line of work culminated in the development of Iris [Jung et al. 2015, 2016; Krebbers et al. 2017a; Jung et al. 2018b]: a unifying, language-generic framework for *higher-order concurrent separation logic*, implemented in the Coq proof assistant [Krebbers et al. 2017b, 2018], into which a variety of SKLR models can be (and have already been) encoded. We give an overview of SKLR models that have been encoded in Iris in §10.

4.4 How Iris Solves the Problems of SKLR Models

Using Iris, the pain points of working with SKLR models—the global, "tedious" reasoning—can largely be made to disappear—replaced by local, "interesting" reasoning. In particular, concerning the complications of working directly with SKLR models that we mentioned in §4.3, Iris addresses them head-on:

(1) **Eliminating tedious step-indexed reasoning:** In Iris, the tedious details of step-index arithmetic are (to a large extent) hidden within the soundness proof of Iris itself, so that proofs done on top of Iris are not cluttered with them. To the limited extent that step-indexed reasoning is necessary (to avoid logical inconsistency), it is handled abstractly—following prior work by Appel et al. [2007]—using the so-called "later" ($\triangleright$) modality, which enables one to assert that a proposition should hold "one step of computation later".

(2) **Local reasoning about state:** In contrast to direct proofs with SKLR models, which involve manipulation of global state, Iris uses separation logic [O'Hearn et al. 2001; Reynolds 2002] to support *local reasoning* about state. Local reasoning makes proofs about stateful code much more pleasant: when proving semantic soundness of an expression e, we need only reason about what happens to the piece of state that e itself manipulates. Moreover, separation logic is a good fit for formalizing semantic soundness because semantic soundness is a compositional property and separation-logic proofs are compositional by construction.

(3) **High-level reasoning about stateful invariants:** Iris extends vanilla separation logic with two logical mechanisms—*impredicative invariants* (a higher-order generalization, first developed in [Svendsen and Birkedal 2014], of the shared resource invariants from O'Hearn's

[12] The technical appendix accompanying [Dreyer et al. 2012] is similarly detail-oriented in this respect.

[13] In contrast, some subsequent papers employing step-indexed proofs, such as [Krishnaswami et al. 2012; Turon et al. 2013b], may seem marginally less cluttered, but that is only because they systematically elide "boring details" related to step-indexing that are quite easy to get wrong.

$$
\begin{aligned}
\tau &::= 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{iProp} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \ldots && \text{(Types)} \\
t, u, P, Q &::= x \mid \lambda x : \tau.\, t \mid t(u) \mid \mathsf{True} \mid \mathsf{False} \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \mid && \text{(Propositional logic)} \\
&\quad\ \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid && \text{(Higher-order logic)} \\
&\quad\ P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \mathrm{wp}_{\mathcal{E}}\, e\, \{\Phi\} \mid && \text{(Separation logic)} \\
&\quad\ \square P \mid \triangleright P \mid \mu x : \tau.\, t \mid {\mid\!\Rrightarrow_{\mathcal{E}}} P \mid \boxed{P}^{\mathcal{N}} \mid \ldots && \text{(Iris-specific connectives)}
\end{aligned}
$$

Fig. 4. Syntax of Iris. (We use P, Q to represent terms of type *iProp*, and t, u to represent arbitrary terms.)

original concurrent separation logic [O'Hearn 2007; Brookes 2007]) and *user-defined ghost state* (the ability to define custom, domain-specific notions of logical resource). Used in tandem, these two mechanisms enable one to express complex invariants on the private state of ADTs and to reason about those invariants at a much higher level of abstraction than is afforded by the possible worlds of SKLRs.

In short, Iris provides an ideal framework for formalizing *logical type soundness—i.e.,* semantic type soundness proofs for richly-typed programming languages encoded in higher-order separation logic. We now proceed to concretely demonstrate the abovementioned benefits of logical type soundness in the context of our example language **MyLang**.

5 DEFINING A LOGICAL RELATION IN IRIS

Figure 4 shows the syntax of Iris, a higher-order logic extended with connectives from separation logic as well as a few other custom modalities which we will present in due course. In this and the next section, we will show, step-by-step, how indeed Iris provides a natural logical language in which semantic type soundness for realistic languages can be formalized.

It is important to note that Iris is a language-generic framework, meaning that it can be instantiated and used to reason about any language defined by a relatively common form of operational semantics. However, for space reasons, we will not attempt to present Iris in its full generality. Rather, to make things concrete, we will instantiate Iris specifically with the language **MyLang** (from §2), and show how one can build a semantic soundness proof for this representative language.

Furthermore, instead of first explaining the design of Iris and then showing how to use it, we will present the features of Iris on-demand, as they arise in defining a *logical relation* (§5.1)—the key ingredient to encoding our semantic typing judgment for the language **MyLang** (§5.2). Then, in §6, we will show how to use this logical relation to prove semantic type soundness, and in §7, we will show how to formalize the idea of "safe encapsulation of unsafe features" that we presented informally in §3.2.

5.1 The Value and Expression Interpretations of Types

As explained in §4.1, proving semantic type soundness for the language **MyLang** involves defining a semantic version of the **MyLang** typing judgment, $\Delta; \Gamma \vDash e : A$. Before defining the general semantic typing relation, we will first define a *logical relation*, which represents semantic typing for *closed* terms. It is then straightforward, as we will see in §5.2, to lift the logical relation on closed terms to a semantic typing relation on open terms using closing substitutions.

The logical relation for **MyLang**, shown in Figure 5, consists of two semantic interpretations of types A—a *value interpretation* $[\![A]\!]_\delta$ and an *expression interpretation* $[\![A]\!]^{\mathsf{e}}_\delta$—which are defined by structural recursion on A. These interpretations describe which values and expressions, respectively,

$$\llbracket A \rrbracket^{\mathsf{e}}_{\delta} \triangleq \lambda\, e.\, \mathsf{wp}\; e \left\{ \llbracket A \rrbracket_{\delta} \right\}$$

$$\begin{aligned}
\llbracket \alpha \rrbracket_{\delta} &\triangleq \delta(\alpha) \\
\llbracket \mathbf{1} \rrbracket_{\delta} &\triangleq \lambda\, v.\, v = () \\
\llbracket \mathbf{2} \rrbracket_{\delta} &\triangleq \lambda\, v.\, v \in \{\mathtt{true}, \mathtt{false}\} \\
\llbracket \mathbf{Z} \rrbracket_{\delta} &\triangleq \lambda\, v.\, v \in \mathbb{Z} \\
\llbracket A_1 \times A_2 \rrbracket_{\delta} &\triangleq \lambda\, v.\, \exists v_1, v_2.\, (v = (v_1, v_2)) * \llbracket A_1 \rrbracket_{\delta}(v_1) * \llbracket A_2 \rrbracket_{\delta}(v_2) \\
\llbracket A_1 + A_2 \rrbracket_{\delta} &\triangleq \lambda\, v.\, \bigvee_{i \in \{1,2\}} \exists w.\, (v = \mathtt{inj}_i\, w) * \llbracket A_i \rrbracket_{\delta}(w) \\
\llbracket A \to B \rrbracket_{\delta} &\triangleq \lambda\, v.\, \Box \left(\forall w.\, \llbracket A \rrbracket_{\delta}(w) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathsf{e}}_{\delta}(v\, w) \right) \\
\llbracket \forall \alpha.\, A \rrbracket_{\delta} &\triangleq \lambda\, v.\, \Box \left(\forall (\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, \llbracket A \rrbracket^{\mathsf{e}}_{\delta, \alpha \mapsto \Psi}(v \langle\rangle) \right) \\
\llbracket \exists \alpha.\, A \rrbracket_{\delta} &\triangleq \lambda\, v.\, \exists (\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, \exists w.\, (v = \mathtt{pack}\langle w \rangle) * \llbracket A \rrbracket_{\delta, \alpha \mapsto \Psi}(w) \\
\llbracket \mu \alpha.\, A \rrbracket_{\delta} &\triangleq \mu\, (\Psi : \mathit{Val} \to \mathit{iProp}_{\Box}).\, \lambda\, v.\, \exists w.\, (v = \mathtt{fold}\, w) * \triangleright \llbracket A \rrbracket_{\delta, \alpha \mapsto \Psi}(w) \\
\llbracket \mathtt{ref}(A) \rrbracket_{\delta} &\triangleq \lambda\, v.\, \exists (\ell : \mathit{Loc}).\, (v = \ell) * \boxed{\exists w.\, \ell \mapsto w * \llbracket A \rrbracket_{\delta}(w)}^{\mathcal{N}_{\mathsf{ty}}.\ell}
\end{aligned}$$

$$\begin{aligned}
\llbracket \emptyset \rrbracket^{\mathsf{c}}_{\delta}(\epsilon) &\triangleq \mathsf{True} \\
\llbracket \Gamma, x : A \rrbracket^{\mathsf{c}}_{\delta}(\vec{v}\, w) &\triangleq \llbracket \Gamma \rrbracket^{\mathsf{c}}_{\delta}(\vec{v}) * \llbracket A \rrbracket_{\delta}(w)
\end{aligned}$$

$$\Delta; \Gamma \vDash e : A \triangleq \Box \left(\forall \delta, \vec{v}.\, \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta) \mathrel{-\!\!*} \llbracket \Gamma \rrbracket^{\mathsf{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket A \rrbracket^{\mathsf{e}}_{\delta}(e[\vec{v}/\vec{x}]) \right)$$

Fig. 5. The expression interpretation $\llbracket\_\rrbracket^{\mathsf{e}}$, value interpretation $\llbracket\_\rrbracket$, typing context interpretation $\llbracket\_\rrbracket^{\mathsf{c}}$, and semantic typing judgment for **MyLang**.

behave like valid inhabitants of the type A. Here, δ is a semantic *environment*, mapping the free type variables of A to their semantic value interpretations—hence, $\llbracket \alpha \rrbracket_{\delta} = \delta(\alpha)$. We will explain the need for this semantic environment when we come to the cases of the logical relation for types that bind variables, namely universal types, existential types, and recursive types. Until then, the reader can simply ignore the δ parameter, since it is otherwise merely threaded through the definition.

Crucially, note that $\llbracket A \rrbracket_{\delta}$ and $\llbracket A \rrbracket^{\mathsf{e}}_{\delta}$ are interpretations of **MyLang** types *in Iris*—*i.e.,* they are simply Iris predicates of type $\mathit{Val} \to \mathit{iProp}$ and $\mathit{Expr} \to \mathit{iProp}$, respectively, where iProp is the type of Iris propositions.[14] We now proceed to explain the definition of these predicates, step by step.

The expression interpretation. The first line of Figure 5 line shows how the expression interpretation is defined in terms of the value interpretation. Intuitively, a closed expression is in the expression interpretation of a type A if it *computes* a result that is in the value interpretation of A. This intuitive idea can be concisely captured using Iris's *weakest precondition* connective:

$$\llbracket A \rrbracket^{\mathsf{e}}_{\delta} \triangleq \lambda\, e.\, \mathsf{wp}\; e \left\{ \llbracket A \rrbracket_{\delta} \right\}$$

Given a postcondition $\Phi : \mathit{Val} \to \mathit{iProp}$, the connective $\mathsf{wp}\; e\; \{\Phi\}$ represents the weakest precondition ensuring that (1) e is safe to execute, and (2) any result value e computes will satisfy Φ. As such, $\llbracket A \rrbracket^{\mathsf{e}}_{\delta}$ precisely expresses that e is safe to execute (*i.e.,* it will not get stuck), and whatever value it evaluates to will be in the value interpretation of the type A.

The remainder of Figure 5 defines the value interpretation of types.

[14] More specifically, these are *persistent* Iris predicates—*i.e.,* their return type is $\mathit{iProp}_{\Box}$. See §6.1 for more about this point.

Ground types. The value interpretation of ground types is exactly what one would expect: the only value of the unit type $\mathbf{1}$ is the unit value $()$, the values of the Boolean type $\mathbf{2}$ are `true` and `false`, and the values of the integer type $\mathbf{Z}$ are the integers $\mathbb{Z}$.

Product and sum types. The value interpretations of product and sum types are similarly straightforward. Values of type $A_1 \times A_2$ should be tuples (v_1, v_2), where v_1 and v_2 should be in the interpretation of A_1 and A_2, respectively. Values of type $A_1 + A_2$ should be either $\texttt{inj}_1\, w$ or $\texttt{inj}_2\, w$, where w should be in the interpretation of A_1 or A_2, respectively.

The reader may wonder why we use separating conjunction $(P * Q)$ in the definition of $[\![A_1 \times A_2]\!]_\delta$ rather than ordinary conjunction $(P \land Q)$—especially because, as it turns out, one *can* replace all occurrences of $*$ in Figure 5 by $\land$ without changing the meaning of the logical relation. The short answer is that, in Iris proofs, separating conjunction is by far the more commonly used form of conjunction, so in cases where $*$ and $\land$ are interchangeable, we use $*$ for uniformity of notation. We will return to this point in more detail in §6.1, but for the moment, the reader can simply think of $*$ as being synonymous with $\land$.

Function types. The value interpretation of the function type $A \to B$ is perhaps the most iconic and familiar case in Figure 5, as some slight variation of it appears in any proof that calls itself a "logical relations" proof. It expresses that a value v inhabits the type $A \to B$ if v maps arguments in $[\![A]\!]_\delta$ to results in $[\![B]\!]^{\mathsf{e}}_\delta$. Note that this definition imposes *no syntactic restriction* on v—it merely insists that, when v is *used* like a function of type $A \to B$ (*i.e.,* when applied to an argument of type A), it *behaves* like a function of type $A \to B$ (*i.e.,* it is safe to execute and returns a result of type B).

There are three technical points of note here.

First, note that we use the *separating implication* connective $(P \mathrel{-\!\!*} Q)$, aka *magic wand*, instead of ordinary implication $(P \Rightarrow Q)$. The reason is simple: since magic wand is the adjoint connective to separating conjunction—*i.e.,* $P * Q \vdash R$ iff $P \vdash Q \mathrel{-\!\!*} R$ (where $\vdash$ is the entailment relation of Iris)—and since in Iris (as noted above) we work mostly with $*$ rather than $\land$, we correspondingly work mostly with $\mathrel{-\!\!*}$ rather than $\Rightarrow$ as well. However, just as with $*$ vs. $\land$, as we detail in §6.1, the distinction between $\mathrel{-\!\!*}$ and $\Rightarrow$ is not important in the context of our logical relation, and the reader can comfortably gloss over it.

Second, note that $[\![A]\!]_\delta$ appears in a *negative* (contravariant) position in the definition of $[\![A \to B]\!]_\delta$. If one attempted to define the logical relation directly as an inductive predicate, this negative occurrence would cause a problem because it would render the inductive generating function non-monotone, so the definition would not be well-founded. However, as mentioned above, the logical relation is in fact defined by structural recursion on its type parameter, so since A is smaller than $A \to B$, the definition is in fact well-founded. (Indeed, it is precisely this function case that necessitates defining $[\![A]\!]_\delta$ by structural recursion on A.)

Lastly, note that the definition of $[\![A \to B]\!]_\delta$ is wrapped in Iris's *persistence modality* $(\square)$; we will defer explanation of this modality until §6.1.

Universal and existential types. The cases we have seen so far exhibit a common pattern. Types are interpreted semantically using the logical connective to which they are associated via the Curry-Howard correspondence: product types by conjunction, sum types by disjunction, and function types by implication. This pattern explains what is "logical" about a logical relation.

The next two cases continue this pattern, interpreting universal and existential types using logical propositions that are universally and existentially quantified, respectively. For those readers familiar with prior work on logical relations—the "reducibility candidates" of Girard [1972], parametricity à la Reynolds [1983], or the logical characterization of parametricity due to Plotkin and Abadi [1993]—these cases should look very familiar. For other readers, some explanation is in order.

Naively, one might expect that since the type variable α in $\forall\alpha.\, A$ (resp. $\exists\alpha.\, A$) represents an unknown syntactic type B, the definition of the logical relation for these types should universally (resp. existentially) quantify over a syntactic type B and then recurse on $A[B/\alpha]$, as follows:

Ill-founded attempt to define logical relation for $\forall\alpha.\, A$ and $\exists\alpha.\, A$:

$$\llbracket \forall\alpha.\, A \rrbracket_\delta \triangleq \lambda\, v.\, \forall(B : \mathit{Type}).\, \llbracket A[B/\alpha] \rrbracket^{\mathsf{e}}_\delta(v\langle\rangle)$$
$$\llbracket \exists\alpha.\, A \rrbracket_\delta \triangleq \lambda\, v.\, \exists(B : \mathit{Type}).\, \exists w.\, (v = \mathsf{pack}\langle w\rangle) * \llbracket A[B/\alpha] \rrbracket_\delta(w)$$

However, as we have noted above, the logical relation is crucially defined by structural recursion on its type parameter, and $A[B/\alpha]$ is not structurally smaller than $\forall\alpha.\, A$ and $\exists\alpha.\, A$. Even if we were to define the logical relation by recursion on the *size* of the type, we would run into the same problem because, thanks to the *impredicativity* of polymorphism in **MyLang**, the quantified type B could be a type of arbitrary size. Hence, this naive definition is not well-founded.

The solution, due originally to Girard [1972] and shown in Figure 5, is instead to define these cases of the logical relation by quantifying over a *semantic type*, rather than a syntactic type B. By "semantic type", we mean an arbitrary element Ψ drawn from the same space to which the value interpretation of types belongs—that is, Ψ is any (persistent) Iris predicate on values, of type $\mathit{Val} \to \mathit{iProp}_\Box$. (We defer discussion of persistence until §6.1.) Once we have quantified over Ψ, we can then recurse over A, interpreting (free) occurrences of α in A using Ψ. This is achieved by extending the semantic environment δ to map α to Ψ. On a purely technical level, it is easy to see that this solves the problem with well-foundedness, since A is structurally smaller than $\forall\alpha.\, A$ and $\exists\alpha.\, A$. It also goes to show why we needed the semantic environment δ around in the first place.

However, if one has not seen Girard [1972]'s method before, one may well wonder how this could possibly work and what ramifications it has. In particular, the space of semantic types includes many value predicates that are not the value interpretation of any syntactic type, so by quantifying over semantic types, does the definition of $\llbracket \forall\alpha.\, A \rrbracket_\delta$ not become too strong, and the definition of $\llbracket \exists\alpha.\, A \rrbracket_\delta$ too weak? The short answer why this works is *parametricity*: in **MyLang**, abstract types α are "really abstract", in the sense that the language provides no way for the client of α to syntactically analyze the type B by which α is implemented (*i.e.,* the type with which α ultimately gets instantiated at runtime). As such, there is no need to require that α be modeled as a syntactic **MyLang** type; it is fine to instead model α as belonging to the larger space of semantic types. Moreover, Girard's method has the major side benefit that it will enable us to establish invariants on the private data representations of existentially-typed ADTs. We will see how this works when we formalize "safe encapsulation" in §7.

Finally, note that, when we quantify over semantic types, we are fundamentally relying on Iris's support for higher-order (in this case, second-order) impredicative quantification.

Recursive types. The value interpretation of recursive types poses yet another challenge. In principle, we would like to say that $\mathtt{fold}\, w$ inhabits the type $\mu\alpha.\, A$ if w inhabits the type $A[\mu\alpha.\, A/\alpha]$ (just as syntactic typing dictates). However, this would mean defining the value interpretation of $\mu\alpha.\, A$ in terms of the value interpretation of the *larger* type $A[\mu\alpha.\, A/\alpha]$, which is not well-founded.

Enter *guarded recursive predicates*: a distinctive feature of Iris which offers a way out of our predicament. In Iris, the *guarded fixed-point* operator $\mu\, x.\, t$ can be used to define recursive predicates without a restriction on the variance of the recursive occurrences of x in t. In return for this flexibility, all recursive occurrences of x must be *guarded*, meaning that they must appear below a *later modality* ($\triangleright$)—*i.e.,* within a term of the form $\triangleright P$. Subject to this restriction, $(\mu\, x.\, t) = t[\mu\, x.\, t/x]$.

So in particular, unrolling the μ in the recursive type case of Figure 5, we obtain the following:[15]

$$[\![\mu\alpha.\, A]\!]_\delta(v) \;=\; \big(\exists w.\, (v = \texttt{fold}\, w) \;*\; \triangleright [\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(w)\big)$$

This is *almost* exactly what we wanted. The only wrinkle here is the presence of the $\triangleright$ modality, which is there to ensure well-foundedness of the guarded fixed-point. Roughly speaking, $\triangleright P$ means that "P will hold in the future after one step of computation". That means that, if we have $\triangleright P$ in our context when proving a weakest precondition $\mathrm{wp}\; e\; \{\Phi\}$, we cannot make use of P right away; but after we have verified that e can safely take a step of reduction to e', the goal reduces to showing $\mathrm{wp}\; e'\; \{\Phi\}$, and at that point we are allowed to strip the $\triangleright$ off of $\triangleright P$ and make use of P in the rest of the proof. For example, returning to the recursive type case: if we know that $[\![\mu\alpha.\, A]\!]_\delta(v)$, then we know that $v = \texttt{fold}\, w$ for some w, but we only know that $[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(w)$ holds "later" (*i.e.,* under a $\triangleright$). As we shall see in §6.7, this "under a later" knowledge about the semantic well-typedness of w is strong enough for us to be able to establish semantic type soundness.

Reference types. Intuitively, values of the reference type $\texttt{ref}(A)$ should be memory locations ℓ at which the value w stored may change over time but must always be of type A. In order to formalize this intuition in Figure 5, we make use of two features of Iris:

- The *points-to connective* $\ell \mapsto v$ (from vanilla separation logic) asserts exclusive *ownership* of location ℓ, along with the knowledge that it currently stores value v. (We will return to the concept of ownership in §6.1.)
- The *invariant assertion* $\boxed{P}^{\mathcal{N}}$ expresses the knowledge that a proposition P holds *invariantly— i.e.,* at all times. Here, $\mathcal{N}$ denotes the *namespace* of the invariant, which is needed to ensure that invariants are not accessed repeatedly in an unsound fashion (see §6.8 for details).

With these connectives in hand, the formal definition of $[\![\texttt{ref}(A)]\!]_\delta$ in Figure 5 is easy to follow: it says that v inhabits the type $\texttt{ref}(A)$ if v is a location ℓ *and* there is an invariant enforcing that ℓ always points to some value w that inhabits the type A. Here, $\mathcal{N}_{\texttt{ty}}.\ell$ is the unique namespace that we use to designate the invariant on location ℓ.

5.2 The Semantic Typing Judgment

We now proceed to define the semantic typing judgment $\Delta; \Gamma \vDash e : A$ for open expressions e, by lifting the expression interpretation using a closing substitution. To do so, we first need to give the *context interpretation* $[\![\Gamma]\!]^{\mathsf{c}}_\delta : \mathit{List}(\mathit{Val}) \to \mathit{iProp}_\Box$ used in the closing substitution. Intuitively, a list of values $\vec{v}$ is in the interpretation $[\![\Gamma]\!]^{\mathsf{c}}_\delta$ if each value in $\vec{v}$ is in the value interpretation $[\![A]\!]_\delta$ for the corresponding type A in Γ. The formal definition is shown in Figure 5.

Now, let Δ, Γ, e, and A be such that all free variables of e are in the domain $\vec{x}$ of Γ, and all free type variables that appear in Γ or A are in Δ. Then, we write $\Delta; \Gamma \vDash e : A$ to express that the expression e is semantically typed at the type A under the contexts Γ and Δ. The typing judgment is defined as a (persistent) relation in the Iris logic as follows:

$$\Delta; \Gamma \vDash e : A \triangleq \Box \begin{pmatrix} \forall \delta, \vec{v}.\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta) \twoheadrightarrow \\ \qquad [\![\Gamma]\!]^{\mathsf{c}}_\delta(\vec{v}) \twoheadrightarrow [\![A]\!]^{\mathsf{e}}_\delta\,(e[\vec{v}/\vec{x}]) \end{pmatrix}$$

It says that e should inhabit the expression interpretation of A under any semantic environment δ and any closing substitution $\vec{v}$ that satisfies the context interpretation $[\![\Gamma]\!]^{\mathsf{c}}_\delta$. One can see this definition as essentially consisting of iterated applications of the function and universal type cases of the logical relation, which serve to abstract e over its typing context.

[15] The proof of this equivalence relies on the μ-equation, together with the fact that $[\![A]\!]_{\delta,\alpha\mapsto[\![B]\!]_\delta} = [\![A[B/\alpha]]\!]_\delta$, which is proved by straightforward induction on the structure of A.

Rules of (affine) bunched implications:

$$\mathsf{True} * P \dashv\vdash P$$
$$P * Q \dashv\vdash Q * P$$
$$(P * Q) * R \dashv\vdash P * (Q * R)$$

$*$-MONO
$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

$-\!*$-INTRO
$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$-\!*$-ELIM
$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

Rules for the persistence modality:

$\Box$-MONO
$$\frac{P \vdash Q}{\Box P \vdash \Box Q}$$

$\Box$-DUP
$$\Box P \dashv\vdash (\Box P * \Box P)$$

$\Box$-ELIM
$$\Box P \vdash P$$

$\Box$-IDEMP
$$\Box P \vdash \Box \Box P$$

$\Box$-AND-SEP
$$(\Box P \wedge Q) \dashv\vdash (\Box P * Q)$$

$\Box$-IMPL-WAND
$$(\Box P \Rightarrow Q) \dashv\vdash (\Box P \mathrel{-\!\!*} Q)$$

$\Box$-SEP
$$\Box(P * Q) \dashv\vdash (\Box P * \Box Q)$$

$\Box$-AND
$$\Box(P \wedge Q) \dashv\vdash (\Box P \wedge \Box Q)$$

$\Box$-OR
$$\Box(P \vee Q) \dashv\vdash (\Box P \vee \Box Q)$$

$\Box$-FORALL
$$\Box(\forall x.\, P) \dashv\vdash (\forall x.\ \Box P)$$

$\Box$-EXISTS
$$\Box(\exists x.\, P) \dashv\vdash (\exists x.\ \Box P)$$

Rules for guarded recursion and the later modality:

μ-UNFOLD
$$\vdash (\mu x.\, t) = t[\mu x.\, t/x]$$

$\triangleright$-MONO
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

$\triangleright$-INTRO
$$P \vdash \triangleright P$$

LÖB
$$\frac{\triangleright P \vdash P}{\vdash P}$$

$\triangleright$-SEP
$$\triangleright(P * Q) \dashv\vdash (\triangleright P * \triangleright Q)$$

$\triangleright$-AND
$$\triangleright(P \wedge Q) \dashv\vdash (\triangleright P \wedge \triangleright Q)$$

$\triangleright$-OR
$$\triangleright(P \vee Q) \dashv\vdash (\triangleright P \vee \triangleright Q)$$

$\triangleright$-FORALL
$$\triangleright(\forall x.\, P) \dashv\vdash (\forall x.\ \triangleright P)$$

EXISTS-$\triangleright$
$$(\exists x.\ \triangleright P) \vdash \triangleright(\exists x.\, P)$$

$\triangleright$-EXISTS
$$\frac{\mathsf{inhabited}(\tau)}{\triangleright(\exists x : \tau.\, P) \vdash (\exists x : \tau.\ \triangleright P)}$$

$\triangleright$-PERS
$$\triangleright(\Box P) \dashv\vdash \Box(\triangleright P)$$

Rules for weakest preconditions:

$$\Phi(v) \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\} \qquad \text{(WP-VAL)}$$
$$\triangleright((e_1 \rightarrow_{\mathsf{pure}} e_2) * \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \qquad \text{(WP-PURE)}$$
$$\mathsf{wp}_{\mathcal{E}}\, e\, \big\{w.\, \mathsf{wp}_{\mathcal{E}}\, K[w]\, \{\Phi\}\big\} \vdash \mathsf{wp}_{\mathcal{E}}\, K[e]\, \{\Phi\} \qquad \text{(WP-BIND)}$$
$$(\forall w.\, \Phi(w) \mathrel{-\!\!*} \Psi(w)) * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Psi\} \qquad \text{(WP-WAND)}$$
$$\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \ell\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{ref}(v)\, \{\Phi\} \qquad \text{(WP-ALLOC)}$$
$$\triangleright(\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\})) \vdash \mathsf{wp}_{\mathcal{E}}\, !\,\ell\, \{\Phi\} \qquad \text{(WP-LOAD)}$$
$$\triangleright(\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, ()\, \{\Phi\})) \vdash \mathsf{wp}_{\mathcal{E}}\, (\ell \leftarrow w)\, \{\Phi\} \qquad \text{(WP-STORE)}$$
$$\triangleright(\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \mathtt{true}\, \{\Phi\})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{CAS}(\ell, v, w)\, \{\Phi\} \qquad \text{(WP-CAS-SUC)}$$
$$\triangleright((v \neq v') * \ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \mathtt{false}\, \{\Phi\})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{CAS}(\ell, v', w)\, \{\Phi\} \qquad \text{(WP-CAS-FAIL)}$$
$$\triangleright(\ell \mapsto n * (\ell \mapsto (n + m) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, n\, \{\Phi\})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{FAA}(\ell, m)\, \{\Phi\} \qquad \text{(WP-FAA)}$$
$$\triangleright(\mathsf{wp}_{\mathcal{E}}\, ()\, \{\Phi\} * \mathsf{wp}_{\top}\, e\, \{v.\, \mathsf{True}\}) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{fork}\, \{e\}\, \{\Phi\} \qquad \text{(WP-FORK)}$$

Fig. 6. Selected rules of the Iris logic. (We let $P \dashv\vdash Q$ be notation for having both $P \vdash Q$ and $Q \vdash P$.)

Rules for invariants:

INV-ALLOC
$$\triangleright P \vdash \mid\!\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-PERSIST
$$\boxed{P}^{\mathcal{N}} \vdash \Box \boxed{P}^{\mathcal{N}}$$

INV-OPEN-UPD
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} \mid\!\Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\triangleright P * Q)) \vdash \mid\!\Rrightarrow_{\mathcal{E}} Q}$$

INV-OPEN-WP
$$\frac{\mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \Big(\triangleright P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\ \triangleright P * \Phi(v)\}\Big) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

Rules for the update modality:

$\mid\!\Rrightarrow$-MONO
$$\frac{P \vdash Q}{\mid\!\Rrightarrow_{\mathcal{E}} P \vdash \mid\!\Rrightarrow_{\mathcal{E}} Q}$$

$\mid\!\Rrightarrow$-INTRO
$$P \vdash \mid\!\Rrightarrow_{\mathcal{E}} P$$

$\mid\!\Rrightarrow$-TRANS
$$\mid\!\Rrightarrow_{\mathcal{E}} \mid\!\Rrightarrow_{\mathcal{E}} P \vdash \mid\!\Rrightarrow_{\mathcal{E}} P$$

$\mid\!\Rrightarrow$-FRAME
$$Q * \mid\!\Rrightarrow_{\mathcal{E}} P \vdash \mid\!\Rrightarrow_{\mathcal{E}} (Q * P)$$

$\mid\!\Rrightarrow$-TIMELESS
$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash \mid\!\Rrightarrow_{\mathcal{E}} P}$$

$\mid\!\Rrightarrow$-WP
$$\mid\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

WP-$\mid\!\Rrightarrow$
$$\mathsf{wp}_{\mathcal{E}}\, e\, \big\{w.\ \mid\!\Rrightarrow_{\mathcal{E}} \Phi(w)\big\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

Fig. 7. Selected rules for invariants and the update modality of the Iris logic.

6 LOGICAL TYPE SOUNDNESS: PROVING SEMANTIC TYPE SOUNDNESS IN IRIS

In this section, we will demonstrate the method of *logical type soundness—i.e.,* proving semantic type soundness within the separation logic framework of Iris. In particular, this will involve proving (in Iris) semantic versions of the typing rules of the language **MyLang**. For instance, we will prove the following semantic typing rule for function application:

$$\frac{\Delta; \Gamma \vDash e_1 : A \to B \qquad \Delta; \Gamma \vDash e_2 : A}{\Delta; \Gamma \vDash e_1\ e_2 : B}$$

Since the semantic typing judgment is an Iris definition, semantic typing rules are simply implications in Iris. For instance, the above inference rule should be read as:

$$\big(\ \Delta; \Gamma \vDash e_1 : A \to B \quad * \quad \Delta; \Gamma \vDash e_2 : A\ \big) \quad \mathrel{-\!\!*} \quad \Delta; \Gamma \vDash e_1\ e_2 : B$$

Semantic typing rules are proven by unfolding the definition of the semantic typing judgment, the expression interpretation, and the value interpretation. Before carrying out these proofs in detail, we first return to a key technical issue that we have thus far glossed over—persistent propositions and the persistence modality $\Box$ (§6.1). We then explain the proof rules for weakest preconditions (§6.2), and how they are used to derive higher-level reasoning principles for the logical relation (§6.3). At that point, we are well equipped to prove the semantic typing rules (§6.4–§6.8). We then prove the *fundamental theorem*—which states that syntactic typing implies semantic typing—and the *adequacy theorem*—which states that closed semantically well-typed terms are indeed safe to execute (§6.9). In §7, we then demonstrate the key benefit of semantic typing—the ability to reason about "safe encapsulation of unsafe features".

Proofs in Iris. For reference, a selection of the Iris proof rules is given in Figure 6 and Figure 7. Since we use Iris as an ambient logic, our proofs are rather different from ordinary proofs in (higher-order) logic—they involve reasoning in separation logic and make use of the various Iris connectives. To allow the reader to get accustomed to the way such proofs are carried out exactly, we visualize many proofs using proof trees. These proof trees make explicit many low-level details, so as to make clear exactly how the Iris rules are used. However, we would like to stress that when Iris proofs are carried out in practice, they are done in Coq using the Iris Proof Mode [Krebbers et al. 2017b, 2018], which takes care of many of the low-level proof steps automatically.

6.1 Persistent vs. Ephemeral Propositions and the □ Modality

In separation logic, propositions may express *exclusive ownership of resources.* We have already seen the prototypical example of such a proposition, namely the *points-to connective*, $\ell \mapsto v$, which denotes exclusive ownership of a location ℓ storing value v. Along with exclusive ownership of a resource typically comes the right to mutate the resource, which may have the effect of invalidating previously valid assertions about the resource. For example, if we can assert $\ell \mapsto 3$, we have the right to update ℓ to 5, after which we can assert $\ell \mapsto 5$, but at that point $\ell \mapsto 3$ no longer holds. Thus, propositions P expressing exclusive ownership are (to use Iris's terminology) *ephemeral*: although P may hold at one point in a program proof, it may cease to hold later on.

There are many propositions, however, that do not assert exclusive ownership of resources; these propositions are (again following Iris's terminology) *persistent*: once they hold, they hold forever. Examples of persistent propositions include pure facts like equality ($t = u$), as well as invariant assertions $\boxed{P}^{\mathcal{N}}$. Although persistent propositions do not offer any exclusive capabilities to their asserters, they do have an advantage over ephemeral propositions, namely that they are *duplicable.* That is, if P is persistent, then $P \dashv\vdash P * P$. Being duplicable is very useful because it means that once P is proven, it represents *freely shareable knowledge*: P can be used repeatedly, as often as needed, in the rest of the proof. For instance, if we need to prove $P \vdash Q * R$, we can reduce this to proving $P \vdash Q$ and $P \vdash R$, which we could not do if P were ephemeral.

The notion of being persistent is expressed in Iris by means of the *persistence modality* □. The purpose of $\square P$ is to say that P holds without depending on any ephemeral propositions. The most important rules for the persistence modality are $\square P \dashv\vdash \square P * \square P$ (rule □-DUP) and $\square P \vdash P$ (rule □-ELIM), which allow one to freely duplicate $\square P$, and finally obtain P when desired. Using the persistence modality, we can formally define the class $\mathit{iProp}_{\square}$ of persistent propositions, and what it means for a proposition P to be persistent (denoted $\mathsf{persistent}(P)$):[16]

$$\mathit{iProp}_{\square} \triangleq \{P : \mathit{iProp} \mid \mathsf{persistent}(P)\}$$
$$\mathsf{persistent}(P) \triangleq P \vdash \square P$$

As usual for □ in modal logic, we have $\square P \vdash \square\square P$ (□-IDEMP), which implies that $\square P$ is persistent regardless of what P is. Furthermore, using the fact that the □ modality commutes with most logical connectives (see Figure 6), we can show that the class of persistent propositions is closed under separating conjunction, conjunction, disjunction, universal quantification, and existential quantification. Lastly, using the fact that □ is monotone (□-MONO), we can derive the following introduction rule, which says that a □ modality can be introduced if the context is persistent:

$$\text{□-INTRO}\quad \frac{P \vdash Q \qquad \mathsf{persistent}(P)}{P \vdash \square Q}$$

Persistent propositions play an important role when defining a logical relation in separation logic. In particular, in the syntactic typing derivation $x : A \vdash e : B$, the assumption that x has type A may

[16] Since Iris does not support subset types, the type $\mathit{iProp}_{\square}$ cannot be defined internally. Instead, since Iris is an open system (formalized as a shallow embedding using the Coq meta logic), one can define the type $\mathit{iProp}_{\square}$ externally and thereby extend Iris's signature of types accordingly. For the purpose of the paper, further details about this construction can be ignored.

be used repeatedly. In establishing the corresponding semantic typing relation $x : A \vDash e : B$, we quantify over a value v, and must then prove that $[\![A]\!](v) \mathrel{-\!\!*} [\![B]\!]^{\mathrm{e}}(e[v/x])$. To prove that implication, we will need to use the assumption $[\![A]\!](v)$ repeatedly, namely for each occurrence of x in e, which we can only do if it is persistent.

Consequently, we have set up the logical relation in Figure 5 so that $[\![A]\!]_\delta(v)$ is persistent by definition.[17] In particular, the value and expression interpretations have the following types:

$$[\![\_]\!]^{\mathrm{e}}_{(\_)} : \mathit{Type} \to (\mathit{Tvar} \rightharpoonup_{\mathrm{fin}} (\mathit{Val} \to \mathit{iProp}_\square)) \to \mathit{Expr} \to \mathit{iProp}$$
$$[\![\_]\!]_{(\_)} : \mathit{Type} \to (\mathit{Tvar} \rightharpoonup_{\mathrm{fin}} (\mathit{Val} \to \mathit{iProp}_\square)) \to \mathit{Val} \to \mathit{iProp}_\square$$

Here, we require the semantic environments δ (over which the interpretations are parameterized) to map type variables to persistent value predicates (*i.e.,* functions from Val to $\mathit{iProp}_\square$), and we require the value interpretation to return a persistent value predicate as well. Most cases of the value interpretation are persistent by construction. The only two that require some “intervention” in order to ensure persistence are the function and universal type cases: in these cases, we enforce persistence by placing the right-hand side of the definition under a $\square$ modality.

Another important property of persistent propositions is $(\square P \wedge Q) \dashv\vdash (\square P * Q)$ (rule □-AND-SEP), which says that ordinary conjunction and separating conjunction coincide when one conjunct is persistent, *i.e.*, we have $P \wedge Q \dashv\vdash P * Q$ if P or Q is persistent. Similarly, ordinary implication $P \Rightarrow Q$ and magic wand $P \mathrel{-\!\!*} Q$ coincide when P is persistent, which follows from the fact that $(\square P \Rightarrow Q) \dashv\vdash (\square P \mathrel{-\!\!*} Q)$ (rule □-IMPL-WAND). As a result, in the definition of the logical relation in Figure 5, the choice between $\wedge$ vs. $*$, and $\Rightarrow$ vs. $\mathrel{-\!\!*}$, is actually irrelevant. Nevertheless, as explained earlier in §5.1, we prefer to stick to $*$ and $\mathrel{-\!\!*}$ in this paper for uniformity of notation (and thus not having to worry about how to associate $*$ and $\wedge$) and because that is what we actually do in Coq.

6.2 Weakest Preconditions

At the heart of the logical relation—in the definition of the expression interpretation $[\![e]\!]^{\mathrm{e}}_\delta$, as shown in Figure 5—we use Iris's connective $\mathrm{wp}\ e\ \{\Phi\}$ for weakest preconditions. Recall from §5.1 that, given a postcondition $\Phi : \mathit{Val} \to \mathit{iProp}$, the connective $\mathrm{wp}\ e\ \{\Phi\}$ represents the weakest precondition ensuring that (1) e is safe to execute, and (2) any result value e computes will satisfy Φ. We will now go over the proof rules for weakest preconditions from Figure 6.

To improve readability, we often write $\mathrm{wp}\ e\ \{w.\ Q\}$ instead of $\mathrm{wp}\ e\ \{\lambda\, w.\ Q\}$, and omit the binder w in the postcondition if we do not say anything about the return value. The occurrences of the later modality ($\triangleright$) and invariant masks ($\mathcal{E}$) in the proof rules in Figure 6 can be ignored for now; we will come back to those at the end of this subsection and in §6.8, respectively.

Pure expressions. The simplest proof rules for weakest preconditions are WP-VAL and WP-PURE. The rule WP-VAL expresses that if an expression is a value v, then proving $\mathrm{wp}\ v\ \{\Phi\}$ can be reduced to proving the postcondition $\Phi(v)$. The rule WP-PURE expresses that if e_1 reduces to e_2 by a pure step (see Figure 3 for the definition of $\to_{\mathrm{pure}}$), then proving $\mathrm{wp}\ e_1\ \{\Phi\}$ can be reduced to proving $\mathrm{wp}\ e_2\ \{\Phi\}$. For example, we can prove $\mathrm{wp}\ (\texttt{if true then}\ ()\ \texttt{else}\ 42(42))\ \{v.\ v = ()\}$ using WP-PURE and WP-VAL consecutively:[18]

[17] Iris can of course also be used to model substructural type systems, in which the value interpretation $[\![A]\!](v)$ will no longer be persistent, although persistence is useful in Iris for a number of other reasons as well. For examples of substructural type systems modeled in Iris see Jung et al. [2018a, 2021]; Jung [2020]; Dang et al. [2020] for the Rust programming language, and Tassarotti et al. [2017]; Hinrichsen et al. [2021] for session types.

[18] Note that, here and in the following text, we explain the proof trees starting with the conclusion and applying inference rules *bottom-up* to reduce it to simpler hypotheses, as one does generally when mechanizing these proofs in Coq.

$$\dfrac{\dfrac{\dfrac{}{\vdash () = ()}\ \text{=-refl}}{\vdash \mathsf{wp}\ ()\ \{v.\, v = ()\}}\ \textsc{wp-val}}{\vdash \mathsf{wp}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ ()\ \texttt{else}\ 42(42))\ \{v.\, v = ()\}}\ \textsc{wp-pure}$$

Note that to prove an Iris proposition P (such as the weakest precondition in the example), we need to derive the entailment True $\vdash P$. When the left-hand side of the entailment is True, we often omit that side—*i.e.,* we write $\vdash P$ instead of True $\vdash P$.

Let us point out that the weakest precondition in this example is logically equivalent to the semantic typing judgment $\vDash (\texttt{if}\ \texttt{true}\ \texttt{then}\ ()\ \texttt{else}\ 42(42)) : \mathbf{1}$. This simple example thus already demonstrates the flexibility of semantic typing—while $\texttt{if}\ \texttt{true}\ \texttt{then}\ ()\ \texttt{else}\ 42(42)$ cannot be typed syntactically due to the syntactically ill-typed subexpression $42(42)$, it *can* be typed semantically because the subexpression $42(42)$ appears in the else-branch, which never gets executed.

Proof composition. Iris provides two important rules to compose proofs of weakest preconditions:

$$\mathsf{wp}\ e\ \big\{w.\, \mathsf{wp}\ K[w]\ \{\Phi\}\big\} \vdash \mathsf{wp}\ K[e]\ \{\Phi\} \qquad (\textsc{wp-bind})$$

$$(\forall w.\, \Phi(w) \mathrel{-\!\!*} \Psi(w)) * \mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{\Psi\} \qquad (\textsc{wp-wand})$$

The rule wp-bind generalizes the sequencing rule of Hoare logic, and the rule wp-wand generalizes the rules of consequence and framing of separation logic. Concretely, the rule wp-bind expresses that proving a weakest precondition for $K[e]$ can be reduced to proving a weakest precondition for e, followed by a weakest precondition for the continuation $K[w]$, where w is the result value of e. The rule wp-wand provides a form of "internal monotonicity", which allows applying a wand in the postcondition of the weakest precondition. This rule is interderivable from the more conventional rules for "external monotonicity" and framing:

$$\overset{\textsc{wp-mono}}{\dfrac{\forall w.\, \Phi(w) \vdash \Psi(w)}{\mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{\Psi\}}} \qquad \overset{\textsc{wp-frame}}{P * \mathsf{wp}\ e\ \{\Phi\} \vdash \mathsf{wp}\ e\ \{w.\, P * \Phi(w)\}}$$

To see these rules in action, let us assume we have some (unknown) expression e, for which we already have in hand a proof of the weakest precondition $\mathsf{wp}\ e\ \{v.\, v = ()\}$. Now, we will reuse that proof to establish the weakest precondition $\mathsf{wp}\ ((\lambda x.\, x)\ e)\ \{v.\, v = ()\}$. This is done as follows:

$$\dfrac{\dfrac{\vdash \mathsf{wp}\ e\ \{v.\, v = ()\} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () = ()}\ \text{=-refl}}{\vdash \mathsf{wp}\ ()\ \{v.\, v = ()\}}\ \textsc{wp-val}}{\vdash \mathsf{wp}\ ((\lambda x.\, x)\ ())\ \{v.\, v = ()\}}\ \textsc{wp-pure}}{\vdash \forall w.\, (w = ()) \mathrel{-\!\!*} \mathsf{wp}\ ((\lambda x.\, x)\ w)\ \{v.\, v = ()\}}\ \forall\text{-intro, } \mathrel{-\!\!*}\text{-}\textsc{intro}\text{, =-subst}}{\vdash \mathsf{wp}\ e\ \big\{w.\, \mathsf{wp}\ ((\lambda x.\, x)\ w)\ \{v.\, v = ()\}\big\}}\ \textsc{wp-wand}}{\vdash \mathsf{wp}\ ((\lambda x.\, x)\ e)\ \{v.\, v = ()\}}\ \textsc{wp-bind}$$

Here, we use rule wp-bind with the call-by-value evaluation context $K \triangleq (\lambda x.\, x)\ [\,]$. This allows us to prove a weakest precondition for e, followed by a weakest precondition for $K[w] = (\lambda x.\, x)\ w$, where w is the result of e. After applying wp-bind, we need to prove a weakest precondition for e, but the postcondition does not match up with the postcondition of the already proven weakest precondition $\mathsf{wp}\ e\ \{v.\, v = ()\}$. We therefore apply the rule wp-wand, after which we proceed using the rules wp-pure and wp-val, similarly to the previous example.

Stateful expressions. The weakest precondition rules for stateful expressions in Figure 6 make use of a combination of the separating conjunction ($*$) and magic wand ($\mathrel{-\!\!*}$) to denote what resources need to be exchanged. For example:

$$(\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} \mathsf{wp}\; v\; \{\Phi\})) \vdash \mathsf{wp}\; !\,\ell\; \{\Phi\} \qquad (\textsc{wp-load})$$

$$(\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} \mathsf{wp}\; ()\; \{\Phi\})) \vdash \mathsf{wp}\; (\ell \leftarrow w)\; \{\Phi\} \qquad (\textsc{wp-store})$$

The rule wp-load for the dereferencing $!\,\ell$ uses the separating conjunction ($*$) to state that ownership of $\ell \mapsto v$ needs to be given up (where v is the value stored at ℓ), and the magic wand ($\mathrel{-\!\!*}$) to state that ownership of $\ell \mapsto v$ is then given back (where the value v remains unchanged) for proving the weakest precondition $\mathsf{wp}\; v\; \{\Phi\}$. Similarly, the rule wp-store for the assignment $\ell \leftarrow w$ states that $\ell \mapsto v$ needs to be given up (where v is the old value stored at ℓ), and $\ell \mapsto w$ is given back (where w is the new value stored at ℓ).

Let us see the rule wp-store in action:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P * \ell \mapsto w \vdash P * \ell \mapsto w}}{P \vdash \ell \mapsto w \mathrel{-\!\!*} (P * \ell \mapsto w)}\;\textsc{-*-intro} \qquad \dfrac{}{\ell \mapsto v \vdash \ell \mapsto v}}{P * \ell \mapsto v \vdash (\ell \mapsto w \mathrel{-\!\!*} (P * \ell \mapsto w)) * \ell \mapsto v}\;*\textsc{-mono}}{P * \ell \mapsto v \vdash \ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} (P * \ell \mapsto w))}\;*\text{-comm}}{P * \ell \mapsto v \vdash \mathsf{wp}\; (\ell \leftarrow w)\; \{P * \ell \mapsto w\}}\;\textsc{wp-store}$$

After applying wp-store, we use $*$-mono to give up the hypothesis $\ell \mapsto v$—this is often called “framing out a hypothesis”—and then use -*-intro to introduce $\ell \mapsto w$. To frame out $\ell \mapsto v$, we use commutativity of the separating conjunction ($*$) so as to ensure $\ell \mapsto v$ appears in the same position on both sides of the entailment relation ($\vdash$). In larger proofs we leave reasoning up to commutativity (and associativity) implicit because it quickly becomes tedious, and in practice, the Iris Proof Mode in Coq [Krebbers et al. 2017b, 2018] takes care of that automatically.

We have previously seen how the rule wp-wand makes it possible to compose proofs of pure expressions. Now let us see how that rule is used for stateful expressions. Suppose we have proved

$$\ell \mapsto n \mathrel{-\!\!*} \mathsf{wp}\; (\mathtt{inc}\; \ell)\; \{w.\, w = () * \ell \mapsto (n+1)\},$$

where $\mathtt{inc} \triangleq \lambda x.\, x \leftarrow (!\,x + 1)$, and we wish to prove

$$(\ell_1 \mapsto n_1 * \ell_2 \mapsto n_2) \mathrel{-\!\!*} \mathsf{wp}\; (\mathtt{inc}\; \ell_1;\; \mathtt{inc}\; \ell_2)\; \{w.\, (w = ()) * \ell_1 \mapsto (n_1 + 1) * \ell_2 \mapsto (n_2 + 1)\}.$$

A proof tree for this example is as follows:

$$\dfrac{\dfrac{\ell_1 \mapsto n_1 \vdash \mathsf{wp}\; \mathtt{inc}\; \ell_1\; \{u.\, u = () * \ell_1 \mapsto (n_1{+}1)\} \qquad \dfrac{\dfrac{\dfrac{\ell_2 \mapsto n_2 \vdash \mathsf{wp}\; \mathtt{inc}\; \ell_2\; \{\ldots\} \qquad \dfrac{\ldots}{\ell_1 \mapsto (n_1{+}1) \vdash \forall w.\, (w = () * \ell_2 \mapsto (n_2{+}1)) \mathrel{-\!\!*} \Phi(w)}}{\ell_2 \mapsto n_2 * \ell_1 \mapsto (n_1{+}1) \vdash \mathsf{wp}\; \mathtt{inc}\; \ell_2\; \{\Phi\}}\;\begin{matrix}\textsc{wp-wand},\\ *\textsc{-mono}\end{matrix}}{\ell_2 \mapsto n_2 * \ell_1 \mapsto (n_1{+}1) \vdash \mathsf{wp}\; ((); \mathtt{inc}\; \ell_2)\; \{\Phi\}}\;\textsc{wp-pure}}{\begin{matrix}\ell_2 \mapsto n_2 \vdash \forall u.\, (u = () * \ell_1 \mapsto (n_1{+}1)) \mathrel{-\!\!*}\\ \mathsf{wp}\; (u; \mathtt{inc}\; \ell_2)\; \{\Phi\}\end{matrix}}\;\begin{matrix}\forall\text{-intro}, \textsc{-*-intro},\\ =\text{-subst}\end{matrix}}{\dfrac{\ell_1 \mapsto n_1 * \ell_2 \mapsto n_2 \vdash \mathsf{wp}\; \mathtt{inc}\; \ell_1\; \{u.\, \mathsf{wp}\; (u; \mathtt{inc}\; \ell_2)\; \{\Phi\}\}}{\ell_1 \mapsto n_1 * \ell_2 \mapsto n_2 \vdash \mathsf{wp}\; (\mathtt{inc}\; \ell_1; \mathtt{inc}\; \ell_2)\; \{\Phi\}}\;\textsc{wp-bind}}\;\begin{matrix}\textsc{wp-wand},\\ *\textsc{-mono}\end{matrix}}{\vdash (\ell_1 \mapsto n_1 * \ell_2 \mapsto n_2) \mathrel{-\!\!*} \mathsf{wp}\; (\mathtt{inc}\; \ell_1; \mathtt{inc}\; \ell_2)\; \{\Phi\}}\;\textsc{-*-intro}$$

Here, we let $\Phi(w) \triangleq (w = ()) * \ell_1 \mapsto (n_1 + 1) * \ell_2 \mapsto (n_2 + 1)$. To use the specification of inc, we first apply wp-wand. The leftmost leaves of the proof tree follow from the specification of inc, while the rightmost leaf is a tautology that follows from trivial separation logic reasoning.

Löb induction and recursive functions. An important feature of Iris is Löb induction:

$$\textsc{Löb}\quad \dfrac{\triangleright P \vdash P}{\vdash P}$$

This rule allows one to reason about recursive computations by a kind of implicit induction on the number of steps they take. Specifically, when proving a goal P, Löb induction allows one to assume $\triangleright P$, which denotes that P will hold one step of computation *later*. Correspondingly, weakest precondition rules for reasoning about expressions that take a step of computation—such as wp-pure and wp-store—contain a later modality $\triangleright$ in the premise because the verification of the rest of the computation (after the first step) need only be valid “later”. As a consequence, after applying such a rule (backwards) in a program proof, the new goal (*i.e.,* the premise of the rule just applied) will have the form $\triangleright Q$. By the rule $\triangleright$-mono, one can strip the $\triangleright$ off both the goal ($\triangleright Q$) and any hypothesis $\triangleright P$ that had been previously introduced by Löb induction. From that point on, the Löb induction hypothesis P can be used freely in the remainder of the proof.

To see Löb induction in action, let us prove a weakest precondition for a trivial program that loops forever $\mathtt{loop} \triangleq \mathtt{rec}\, f(x) = f\, x$. The specification is $\mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\}$, where the postcondition is False because the program never returns a value. A proof tree for this example is:

$$\cfrac{\cfrac{\overline{\triangleright \mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\} \vdash \triangleright \mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\}}}{\triangleright \mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\} \vdash \mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\}}\ \textsc{wp-pure}}{\vdash \mathsf{wp}\ \mathtt{loop}\ ()\ \{\mathsf{False}\}}\ \textsc{Löb}$$

The Löb rule provides the goal under a later as a hypothesis. By taking a step of computation ($\mathtt{loop}\ () \rightarrow_{\mathrm{pure}} \mathtt{loop}\ ()$) using wp-pure, we obtain a new goal wrapped in a $\triangleright$ modality, which can then be discharged using the Löb induction hypothesis.

It is important to note that the later modalities $\triangleright$ in the rules for weakest preconditions (Figure 6) make these rules strictly stronger. The later modalities signify that a step of computation has been taken, and thereby make it possible to strip a later off all hypotheses, and in particular the Löb induction hypothesis, as we have seen in the proof above. If it is not needed to strip off a later, versions of the weakest precondition rules without the $\triangleright$ can be derived using the rule $\triangleright$-intro.

6.3 Monadic Rules for the Expression Interpretation

To prove the semantic typing rules in the coming sections (§6.4–§6.8), we typically proceed by unfolding the definition of the semantic typing judgment $\Delta; \Gamma \vDash e : A$, the expression interpretation $[\![A]\!]^{\mathsf{e}}_{\delta}(e)$, and the value interpretation $[\![A]\!]_{\delta}(v)$, to obtain an Iris proposition that we then prove using Iris’s proof rules. To streamline these proofs, we prove the following auxiliary rules, whose statements resemble the types of the monadic operators return and bind.

Lemma 6.1 (The monadic rules for the expression interpretation).

$$[\![A]\!]_{\delta}(v) \mathrel{-\!\!*} [\![A]\!]^{\mathsf{e}}_{\delta}(v) \qquad \text{(logrel-val)}$$

$$[\![A]\!]^{\mathsf{e}}_{\delta}(e) * \big(\forall v.\, [\![A]\!]_{\delta}(v) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_{\delta}(K[v])\big) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_{\delta}(K[e]) \qquad \text{(logrel-bind)}$$

Proof. The rule logrel-val follows by unfolding the expression interpretation and the rule wp-val. The rule logrel-bind follows by unfolding the expression interpretation and a combination of wp-bind and wp-wand. The proof is visualized in the following proof tree:

$$\cfrac{\cfrac{\overline{\mathsf{wp}\ e\ \{[\![A]\!]_{\delta}\} * (\forall v.\, [\![A]\!]_{\delta}(v) \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{[\![B]\!]_{\delta}\}) \vdash \mathsf{wp}\ e\ \{v.\, \mathsf{wp}\ K[v]\ \{[\![B]\!]_{\delta}\}\}}\ \textsc{wp-wand}}{\mathsf{wp}\ e\ \{[\![A]\!]_{\delta}\} * (\forall v.\, [\![A]\!]_{\delta}(v) \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{[\![B]\!]_{\delta}\}) \vdash \mathsf{wp}\ K[e]\ \{[\![B]\!]_{\delta}\}}\ \textsc{wp-bind}}{[\![A]\!]^{\mathsf{e}}_{\delta}(e) * (\forall v.\, [\![A]\!]_{\delta}(v) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_{\delta}(K[v])) \vdash [\![B]\!]^{\mathsf{e}}_{\delta}(K[e])}\ \text{unfold } [\![A]\!]^{\mathsf{e}} \text{ and } [\![B]\!]^{\mathsf{e}} \text{ on LHS/RHS}$$

□

The logrel-bind rule is particularly useful in the proofs of semantic typing rules because it enables us to “zap” an unknown but semantically well-typed term to a semantically well-typed value and proceed with the proof. Specifically, suppose we are trying to establish a goal of the form $\llbracket A \rrbracket^{\mathrm{e}}_{\delta}(e) * P \vdash \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(K[e])$. That is, we want to prove that $K[e]$ is semantically well-typed at type B, and we have by assumption that the first subexpression to be evaluated (e) is semantically well-typed at type A. Now if we knew what e was, then we could proceed by evaluating it, but often when proving semantic typing rules, we *don't* know what e is, so it may seem the proof is stuck (*i.e.,* it is universally quantified by the typing rule). Fortunately, in these cases, we can instead apply the LOGREL-BIND rule to reduce the goal to one in which the occurrences of the unknown e are "zapped" to (*i.e.,* replaced by) an unknown value v, as follows:

$$\dfrac{\dfrac{\dfrac{\llbracket A \rrbracket_{\delta}(v) * P \vdash \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(K[v])}{P \vdash \forall v.\, \llbracket A \rrbracket_{\delta}(v) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(K[v])}\;\forall\text{-intro}, \mathrel{-\!\!*}\text{-INTRO}}{\llbracket A \rrbracket^{\mathrm{e}}_{\delta}(e) * P \vdash \llbracket A \rrbracket^{\mathrm{e}}_{\delta}(e) * (\forall v.\, \llbracket A \rrbracket_{\delta}(v) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(K[v]))}\;*\text{-MONO}}{\llbracket A \rrbracket^{\mathrm{e}}_{\delta}(e) * P \vdash \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(K[e])}\;\text{LOGREL-BIND}$$

We can then proceed by unfolding the definition of $\llbracket A \rrbracket_{\delta}(v)$, which typically yields information about v that allows us to make progress in evaluating $K[v]$.

In the following sections, we will use the above proof pattern repeatedly and refer to it simply as "applying the LOGREL-BIND rule".

6.4 Variables and Ground Types

We are now ready to start proving semantic versions of the typing rules of the language **MyLang**. Let us begin with the rules for variables and ground types (unit, Boolean, and integer).

Proof of S-VAR. With the definition of semantic typing and some basic results at hand, we can now state the first semantic typing rule for our language:

$$\textsc{S-var}\quad \dfrac{x : A \in \Gamma}{\Delta; \Gamma \vDash x : A}$$

The proof of this typing rule follows almost immediately from the way we defined the semantic typing judgment. By unfolding the definition of the judgment $\Delta; \Gamma \vDash x : A$, we see that it amounts to proving $\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket A \rrbracket^{\mathrm{e}}_{\delta}\,(x[\vec{v}/\vec{x}])$ for any $\vec{v}$ and δ with $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta)$. From $x : A \in \Gamma$, we obtain that $\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v})$ entails $\llbracket A \rrbracket_{\delta}\,(v)$. The expression interpretation $\llbracket A \rrbracket^{\mathrm{e}}_{\delta}\,(x[\vec{v}/\vec{x}])$ can be reduced to $\llbracket A \rrbracket^{\mathrm{e}}_{\delta}\,(v)$. The proof can be concluded using rule LOGREL-VAL from Lemma 6.1.

Let us proceed with the ground types, whose value interpretations we recall from Figure 5:

$$\llbracket \mathbf{1} \rrbracket_{\delta} \triangleq \lambda v.\, v = () \qquad \llbracket \mathbf{2} \rrbracket_{\delta} \triangleq \lambda v.\, v \in \{\mathtt{true}, \mathtt{false}\} \qquad \llbracket \mathbf{Z} \rrbracket_{\delta} \triangleq \lambda v.\, v \in \mathbb{Z}$$

As explained in §5.1, these interpretations are exactly what one would expect: the only value of the unit type **1** is the unit value $()$, the values of the Boolean type **2** are `true` and `false`, and the values of the integer type **Z** are the integer literals $\mathbb{Z}$. The semantic typing rules for introduction of these types are as follows:

$$\textsc{S-unit}\quad \Delta; \Gamma \vDash () : \mathbf{1} \qquad\qquad \textsc{S-int}\quad \dfrac{n \in \mathbb{Z}}{\Delta; \Gamma \vDash n : \mathbf{Z}} \qquad\qquad \textsc{S-bool}\quad \dfrac{b \in \{\mathtt{true}, \mathtt{false}\}}{\Delta; \Gamma \vDash b : \mathbf{2}}$$

These semantic typing rules are proven by unfolding the definition of the semantic typing judgment and making use of the rule LOGREL-VAL. For example, $\Delta; \Gamma \vDash () : \mathbf{1}$ unfolds to $\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket \mathbf{1} \rrbracket^{\mathrm{e}}_{\delta}\,(()[\vec{v}/\vec{x}])$ for any $\vec{v}$ and δ with $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta)$. The expression interpretation $\llbracket \mathbf{1} \rrbracket^{\mathrm{e}}_{\delta}\,(()[\vec{v}/\vec{x}])$ can be reduced to the value interpretation $\llbracket \mathbf{1} \rrbracket_{\delta}()$ by rule LOGREL-VAL. The value interpretation $\llbracket \mathbf{1} \rrbracket_{\delta}()$, in turn, can be reduced to the tautology $() = ()$.

While the proofs of the preceding rules almost immediately follow from unfolding the definition of the typing judgment, the following rules are more interesting to prove:

$$\textsc{S-if}\quad \dfrac{\Delta;\Gamma \vDash e : \mathbf{2} \qquad \Delta;\Gamma \vDash e_1 : B \qquad \Delta;\Gamma \vDash e_2 : B}{\Delta;\Gamma \vDash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : B} \qquad\qquad \textsc{S-fork}\quad \dfrac{\Delta;\Gamma \vDash e : A}{\Delta;\Gamma \vDash \mathtt{fork}\ \{e\} : \mathbf{1}}$$

Proof of S-IF. In order to prove the rule S-IF, we prove the following auxiliary result for closed expressions, from which we later prove the semantic typing rule:

$$\llbracket \mathbf{2} \rrbracket^{\mathrm{e}}_{\delta}(e) * \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_1) * \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_2) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{wp}\ e_1\ \{\llbracket B\rrbracket_\delta\} * \mathsf{wp}\ e_2\ \{\llbracket B\rrbracket_\delta\} \vdash \mathsf{wp}\ e_1\ \{\llbracket B\rrbracket_\delta\}}}{\mathsf{wp}\ e_1\ \{\llbracket B\rrbracket_\delta\} * \mathsf{wp}\ e_2\ \{\llbracket B\rrbracket_\delta\} \vdash \mathsf{wp}\ \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{\llbracket B\rrbracket_\delta\}}\ \textsc{wp-pure} \qquad \dfrac{\dots}{\text{case for } \mathtt{false}}}{v \in \{\mathtt{true}, \mathtt{false}\} * \mathsf{wp}\ e_1\ \{\llbracket B\rrbracket_\delta\} * \mathsf{wp}\ e_2\ \{\llbracket B\rrbracket_\delta\} \vdash \mathsf{wp}\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{\llbracket B\rrbracket_\delta\}}}{\llbracket \mathbf{2}\rrbracket_\delta(v) * \mathsf{wp}\ e_1\ \{\llbracket B\rrbracket_\delta\} * \mathsf{wp}\ e_2\ \{\llbracket B\rrbracket_\delta\} \vdash \mathsf{wp}\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \{\llbracket B\rrbracket_\delta\}}\ \text{unfold } \llbracket \mathbf{2}\rrbracket \text{ on LHS}}{\llbracket \mathbf{2}\rrbracket_\delta(v) * \llbracket B\rrbracket^{\mathrm{e}}_\delta(e_1) * \llbracket B\rrbracket^{\mathrm{e}}_\delta(e_2) \vdash \llbracket B\rrbracket^{\mathrm{e}}_\delta(\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)}\ \text{unfold } \llbracket B\rrbracket^{\mathrm{e}} \text{ on LHS/RHS}}{\llbracket \mathbf{2}\rrbracket^{\mathrm{e}}_\delta(e) * \llbracket B\rrbracket^{\mathrm{e}}_\delta(e_1) * \llbracket B\rrbracket^{\mathrm{e}}_\delta(e_2) \vdash \llbracket B\rrbracket^{\mathrm{e}}_\delta(\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)}\ \textsc{logrel-bind}$$

Reading this proof tree bottom-up, we first apply the rule LOGREL-BIND (as discussed in §6.3) with evaluation context $K \triangleq \mathtt{if}\ [\,]\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ to turn the premise $\llbracket \mathbf{2} \rrbracket^{\mathrm{e}}_{\delta}(e)$ (of the expression interpretation) into $\llbracket \mathbf{2} \rrbracket_{\delta}(v)$ (of the value interpretation) for the result v of e. After that, we unfold the definitions of $\llbracket \mathbf{2} \rrbracket$ and $\llbracket B \rrbracket^{\mathrm{e}}$, and perform a case analysis on $v \in \{\mathtt{true}, \mathtt{false}\}$. Both subproofs follow from reasoning using Iris's rule WP-PURE.

To prove the actual semantic typing rule S-IF, we unfold the definition of the semantic typing judgment $\Delta;\Gamma \vDash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : B$, which shows that we have to prove that:

$$\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}\,(\mathtt{if}\ e[\vec{v}/\vec{x}]\ \mathtt{then}\ e_1[\vec{v}/\vec{x}]\ \mathtt{else}\ e_2[\vec{v}/\vec{x}])$$

follows from the assumptions $\Delta;\Gamma \vDash e : \mathbf{2}$ and $\Delta;\Gamma \vDash e_1 : B$ and $\Delta;\Gamma \vDash e_2 : B$, which unfold to:

$$\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket \mathbf{2} \rrbracket^{\mathrm{e}}_{\delta}(e[\vec{v}/\vec{x}]) \quad \text{and} \quad \llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_1[\vec{v}/\vec{x}]) \quad \text{and} \quad \llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_2[\vec{v}/\vec{x}]).$$

Since the interpretation $\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v})$ of typing contexts is persistent, we can duplicate it. Our goal then follows from the auxiliary result $\llbracket \mathbf{2} \rrbracket^{\mathrm{e}}_{\delta}(e) * \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_1) * \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(e_2) \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_{\delta}(\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)$ for closed expressions that we proved above.

A note about proofs. If the reader finds the proof for S-IF above to be surprisingly long, we would like to emphasize that this is only because we include many details for didactic reasons. Once one has some experience with proofs in Iris, many details can be elided; and in fact, we will start doing so throughout this paper. The mechanized proof of S-IF in Coq, for example, is only 4 lines long.

Proof of S-FORK. To prove this rule, we first prove the following auxiliary result for closed expressions, from which the semantic typing rule follows immediately:

$$\llbracket A \rrbracket^{\mathrm{e}}_{\delta}(e) \mathrel{-\!\!*} \llbracket \mathbf{1} \rrbracket^{\mathrm{e}}_{\delta}(\mathtt{fork}\ \{e\})$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash () = ()}\,\text{=-refl}}{\vdash [\![\mathbf{1}]\!]_\delta()}\,\text{unfold } [\![\mathbf{1}]\!] \text{ on RHS}}{\vdash \mathsf{wp}\ ()\ \{[\![\mathbf{1}]\!]_\delta\}}\,\textsc{wp-val} \qquad \dfrac{\dfrac{}{\vdash \forall v.\ [\![A]\!]_\delta(v) \mathrel{-\!\!*} \mathsf{True}}}{\mathsf{wp}\ e\ \{[\![A]\!]_\delta\} \vdash \mathsf{wp}\ e\ \{\mathsf{True}\}}\,\textsc{wp-wand}}{\mathsf{wp}\ e\ \{[\![A]\!]_\delta\} \vdash \mathsf{wp}\ ()\ \{[\![\mathbf{1}]\!]_\delta\} * \mathsf{wp}\ e\ \{\mathsf{True}\}}\,{*}\textsc{-mono}}{\mathsf{wp}\ e\ \{[\![A]\!]_\delta\} \vdash \mathsf{wp}\ \mathtt{fork}\ \{e\}\ \{[\![\mathbf{1}]\!]_\delta\}}\,\textsc{wp-fork}}{[\![A]\!]^{\mathsf{e}}_\delta(e) \vdash [\![\mathbf{1}]\!]^{\mathsf{e}}_\delta(\mathtt{fork}\ \{e\})}\,\text{unfold } [\![A]\!]^{\mathsf{e}} \text{ on LHS, unfold } [\![\mathbf{1}]\!]^{\mathsf{e}} \text{ on RHS}$$

The key part of this proof is the use of Iris's rule for fork:

$$\triangleright(\mathsf{wp}\ ()\ \{\Phi\} * \mathsf{wp}\ e\ \{\mathsf{True}\}) \vdash \mathsf{wp}\ \mathtt{fork}\ \{e\}\ \{\Phi\} \qquad (\textsc{wp-fork})$$

This rule says that to prove a weakest precondition for $\mathtt{fork}\ \{e\}$, we need to prove a weakest precondition $\mathsf{wp}\ ()\ \{\Phi\}$ for the main thread separately from a weakest precondition $\mathsf{wp}\ e\ \{\mathsf{True}\}$ for the forked-off thread. Note that a forked-off expression is allowed to return any value since its result is thrown away, hence the postcondition is simply True.

It is worth noting that neither the proof of S-fork, nor the proof of any other semantic typing rule, involves explicit reasoning about the thread-pool semantics. This kind of reasoning is hidden by working in the Iris logic.

6.5 Product, Sum, and Function Types

Recall from Figure 5 the value interpretation for product, sum, and function types:

$$[\![A_1 \times A_2]\!]_\delta \triangleq \lambda v.\ \exists v_1, v_2.\ (v = (v_1, v_2)) * [\![A_1]\!]_\delta(v_1) * [\![A_2]\!]_\delta(v_2)$$
$$[\![A_1 + A_2]\!]_\delta \triangleq \lambda v.\ \bigvee_{i\in\{1,2\}} \exists w.\ (v = \mathtt{inj}_i\ w) * [\![A_i]\!]_\delta(w)$$
$$[\![A \to B]\!]_\delta \triangleq \lambda v.\ \Box\left(\forall w.\ [\![A]\!]_\delta(w) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(v\ w)\right)$$

As explained in §5.1, values of $A_1 \times A_2$ are tuples (v_1, v_2), where v_1 and v_2 are in the interpretation of A_1 and A_2, respectively. Values of $A_1 + A_2$ are either $\mathtt{inj}_1\ w$ or $\mathtt{inj}_2\ w$, where w is in the interpretation of A_1 or A_2, respectively. Values of $A \to B$ are functions v that map arguments in the interpretation of A to results $v\ w$ in the interpretation of B. Recall from §6.1 that the $\Box$ modality is used to enforce that the interpretation of the function type $A \to B$ is persistent.

For products and sums, we prove semantic typing rules corresponding to the syntactic typing rules T-pair, T-proj, T-inj, T-match-sum. The proofs of these rules proceed in a similar way as the proofs we have seen in §6.4. More interesting are the rules for functions:

$$\overset{\textsc{S-app}}{\dfrac{\Delta; \Gamma \vDash e_1 : A \to B \qquad \Delta; \Gamma \vDash e_2 : A}{\Delta; \Gamma \vDash e_1\ e_2 : B}} \qquad\qquad \overset{\textsc{S-rec}}{\dfrac{\Delta; \Gamma, x : A, f : A \to B \vDash e : B}{\Delta; \Gamma \vDash \mathtt{rec}\ f(x) = e : A \to B}}$$

Proof of S-app. Similar to the proofs in §6.4, we first prove an following auxiliary result for closed expressions, from which the semantic typing rule S-app follows immediately:

$$[\![A \to B]\!]^{\mathsf{e}}_\delta(e_1) * [\![A]\!]^{\mathsf{e}}_\delta(e_2) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(e_1\ e_2)$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\left([\![A]\!]_\delta(v_2) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(v_1\ v_2)\right) * [\![A]\!]_\delta(v_2) \vdash [\![B]\!]^{\mathsf{e}}_\delta(v_1\ v_2)}\,{-\!\!*}\textsc{-elim} \text{ on LHS}}{\Box\left(\forall w.\ [\![A]\!]_\delta(w) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(v_1\ w)\right) * [\![A]\!]_\delta(v_2) \vdash [\![B]\!]^{\mathsf{e}}_\delta(v_1\ v_2)}\,\Box\textsc{-elim}, \forall\text{-elim on LHS}}{[\![A \to B]\!]_\delta(v_1) * [\![A]\!]_\delta(v_2) \vdash [\![B]\!]^{\mathsf{e}}_\delta(v_1\ v_2)}\,\text{unfold } [\![A \to B]\!]^{\mathsf{e}} \text{ on LHS}}{[\![A \to B]\!]_\delta(v_1) * [\![A]\!]^{\mathsf{e}}_\delta(e_2) \vdash [\![B]\!]^{\mathsf{e}}_\delta(v_1\ e_2)}\,\textsc{logrel-bind}}{[\![A \to B]\!]^{\mathsf{e}}_\delta(e_1) * [\![A]\!]^{\mathsf{e}}_\delta(e_2) \vdash [\![B]\!]^{\mathsf{e}}_\delta(e_1\ e_2)}\,\textsc{logrel-bind}$$

Reading this proof tree bottom-up, we start by using LOGREL-BIND twice (following the scheme we described in §6.4), first for expression e_1 in context $K \triangleq [\,]\ e_2$, and then for expression e_2 in context $K \triangleq v_1\ [\,]$. The last step truly demonstrates why “logical relations” are called “logical”—we use Iris’s modus ponens rule $(Q \mathrel{-\!\!*} R) * Q \vdash R$ ($\mathrel{-\!\!*}$-ELIM) to eliminate the magic wand that appears in the interpretation of the function type $A \rightarrow B$.

Proof of S-REC. As usual, we first prove an auxiliary result for closed expressions:

$$\Box\left(\forall w\, v.\ \llbracket A\rrbracket_\delta(w) * \llbracket A \rightarrow B\rrbracket_\delta(v) \mathrel{-\!\!*} \llbracket B\rrbracket^{\mathsf{e}}_\delta(e[w/x][v/f])\right) \mathrel{-\!\!*} \llbracket A \rightarrow B\rrbracket^{\mathsf{e}}_\delta(\mathtt{rec}\, f(x) = e)$$

This result says that $\mathtt{rec}\ f(x) = e$ is in the interpretation of $A \rightarrow B$ if for all values w in the interpretation of A, and for all values v in the interpretation of the recursive call of $A \rightarrow B$, we have that $e[w/x][v/f]$ in the interpretation of $A \rightarrow B$.

Let us abbreviate $P \triangleq \Box\left(\forall w\, v.\ \llbracket A\rrbracket_\delta(w) * \llbracket A \rightarrow B\rrbracket_\delta(v) \mathrel{-\!\!*} \llbracket B\rrbracket^{\mathsf{e}}_\delta(e[w/x][v/f])\right)$. The proof of the auxiliary result is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\llbracket B\rrbracket^{\mathsf{e}}_\delta(e[w/x][\mathtt{rec}\, f(x) = e/f]) \vdash \mathsf{wp}\ e[w/x][\mathtt{rec}\, f(x) = e/f]\ \{\llbracket B\rrbracket_\delta\}}\ \text{unfold } \llbracket B\rrbracket^{\mathsf{e}} \text{ on LHS}}{P * \llbracket A\rrbracket_\delta(w) * \llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e) \vdash \mathsf{wp}\ e[w/x][\mathtt{rec}\, f(x) = e/f]\ \{\llbracket B\rrbracket_\delta\}}\ \Box\text{-ELIM, }\forall\text{-elim, } \mathrel{-\!\!*}\text{-ELIM in } P}{P * \llbracket A\rrbracket_\delta(w) * \triangleright\llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e) \vdash \triangleright\,\mathsf{wp}\ e[w/x][\mathtt{rec}\, f(x) = e/f]\ \{\llbracket B\rrbracket_\delta\}}\ \triangleright\text{-MONO}}{P * \llbracket A\rrbracket_\delta(w) * \triangleright\llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e) \vdash \mathsf{wp}\ (\mathtt{rec}\, f(x) = e)\ w\ \{\llbracket B\rrbracket_\delta\}}\ \text{WP-PURE}}{P * \triangleright\llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e) \vdash \Box\left(\forall w.\ \llbracket A\rrbracket_\delta(w) \mathrel{-\!\!*} \mathsf{wp}\ (\mathtt{rec}\, f(x) = e)\ w\ \{\llbracket B\rrbracket_\delta\}\right)}\ \begin{array}{l}\Box\text{-INTRO, }\forall\text{-intro,}\\ \mathrel{-\!\!*}\text{-INTRO}\end{array}}{P * \triangleright\llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e) \vdash \llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e)}\ \begin{array}{l}\text{unfold } \llbracket A \rightarrow B\rrbracket\\ \text{and } \llbracket B\rrbracket^{\mathsf{e}} \text{ on RHS}\end{array}}{P \vdash \llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\, f(x) = e)}\ \text{LÖB}}{P \vdash \llbracket A \rightarrow B\rrbracket^{\mathsf{e}}_\delta(\mathtt{rec}\, f(x) = e)}\ \text{unfold } \llbracket A \rightarrow B\rrbracket^{\mathsf{e}} \text{ on RHS, WP-VAL}$$

The key step of this proof is the use of the rule LÖB for Löb induction, using which we obtain the induction hypothesis (IH) $\triangleright\llbracket A \rightarrow B\rrbracket_\delta(\mathtt{rec}\ f(x) = e)$. Subsequently, we proceed by unfolding the value interpretation of the function type $A \rightarrow B$, after which we obtain the resulting goal $\Box\left(\forall w.\ \llbracket A\rrbracket_\delta(w) \mathrel{-\!\!*} \mathsf{wp}\ (\mathtt{rec}\ f(x) = e)\ w\ \{\llbracket B\rrbracket_\delta\}\right)$. We then introduce the $\Box$ modality, universal quantifier, and magic wand, and use WP-PURE to reduce $(\mathtt{rec}\ f(x) = e)\ w$ to $e[w/x][v/f]$ by performing a step of computation. As a result of that, we obtain a later modality $\triangleright$ in our goal, allowing us to use $\triangleright$-MONO to obtain the IH *now* (*i.e.*, without $\triangleright$). We then eliminate the magic wand connectives in the premise $P \triangleq \Box\left(\forall w\, v.\ \llbracket A\rrbracket_\delta(w) * \llbracket A \rightarrow B\rrbracket_\delta(v) \mathrel{-\!\!*} \llbracket B\rrbracket^{\mathsf{e}}_\delta(e[w/x][v/f])\right)$ to obtain $\llbracket B\rrbracket^{\mathsf{e}}_\delta(e[w/x][\mathtt{rec}\ f(x) = e/f])$, which matches our goal exactly.

A formal note. The primitive version of Iris’s rule LÖB is restricted to the empty context (*i.e.*, the LHS of the entailment $\vdash$ should be True). However, in the above proof, the context is non-empty (it contains P). We therefore in fact use the following derived rule:

$$\frac{P * \triangleright Q \vdash Q \qquad \mathrm{persistent}(P)}{P \vdash Q}$$

6.6 Universal and Existential Types

Recall from Figure 5 the value interpretation for universal and existential types:

$$\begin{aligned}\llbracket \alpha\rrbracket_\delta &\triangleq \delta(\alpha)\\ \llbracket \forall\alpha.\ A\rrbracket_\delta &\triangleq \lambda v.\ \Box\left(\forall(\Psi : \mathit{Val} \rightarrow \mathit{iProp}_\Box).\ \llbracket A\rrbracket^{\mathsf{e}}_{\delta,\alpha\mapsto\Psi}(v\langle\rangle)\right)\\ \llbracket \exists\alpha.\ A\rrbracket_\delta &\triangleq \lambda v.\ \exists(\Psi : \mathit{Val} \rightarrow \mathit{iProp}_\Box).\ \exists w.\ (v = \mathtt{pack}\langle w\rangle) * \llbracket A\rrbracket_{\delta,\alpha\mapsto\Psi}(w)\end{aligned}$$

As explained in §5.1, the semantic environment δ maps the free type variables to their semantic value interpretations—hence, $\llbracket \alpha\rrbracket_\delta = \delta(\alpha)$. The value interpretation of $\forall\alpha.\ A$ and $\exists\alpha.\ A$ quantify over a semantic type $\Psi : \mathit{Val} \to \mathit{iProp}_{\square}$ using Iris's universal and existential quantifier, respectively. Within the quantification, they extend the semantic environment δ of the value interpretation of A to map α to Ψ. Note that since the expression $\llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(v\langle\rangle)$ is not persistent (it is defined in terms of a weakest precondition, which is not persistent), we wrap the value interpretation of $\forall\alpha.\, A$ in a persistence modality $\square$ to ensure it is persistent.

The proofs of the semantic typing rules corresponding to T-tapp, T-tlam, T-pack, and T-match-ex crucially rely on Iris's rules for quantifiers. We additionally need the following lemma, which says that substitution in types corresponds to extending the semantic type environment.

Lemma 6.2. $\llbracket A[B/\alpha] \rrbracket_{\delta} = \llbracket A \rrbracket_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}$ *and* $\llbracket A[B/\alpha] \rrbracket^{\mathrm{e}}_{\delta} = \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}$.

Proof. The first equality is proven by induction on the structure of A, the second is a trivial consequence of the first by the definition of the expression interpretation $\llbracket \_ \rrbracket^{\mathrm{e}}$. □

Let us show the proofs for the elimination and introduction rules for universal types:

$$\text{S-tapp}\quad \frac{\Delta;\Gamma \vDash e : \forall\alpha.\, A \qquad \mathrm{FV}(B) \subseteq \Delta}{\Delta;\Gamma \vDash e\langle\rangle : A[B/\alpha]} \qquad\qquad \text{S-tlam}\quad \frac{\Delta,\alpha;\Gamma \vDash e : A}{\Delta;\Gamma \vDash \Lambda.\, e : \forall\alpha.\, A}$$

Proof of S-tapp. Following the usual approach, to prove the semantic typing rule, we first prove an auxiliary result for closed expressions:

$$\llbracket \forall\alpha.\, A \rrbracket^{\mathrm{e}}_{\delta}(e) \mathrel{-\!\!*} \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}(e\langle\rangle)$$

This auxiliary result is proved as follows:

$$\begin{array}{rl}
\hline
\square\big(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_{\square}).\, \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(v\langle\rangle)\big) \vdash \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}(v\langle\rangle) & \square\text{-elim, }\forall\text{-elim on LHS} \\
\hline
\llbracket \forall\alpha.\, A \rrbracket_{\delta}(v) \vdash \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}(v\langle\rangle) & \text{unfold } \llbracket \forall\alpha.\, A \rrbracket^{\mathrm{e}} \text{ on LHS} \\
\hline
\llbracket \forall\alpha.\, A \rrbracket^{\mathrm{e}}_{\delta}(e) \vdash \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\llbracket B \rrbracket_{\delta}}(e\langle\rangle) & \text{logrel-bind}
\end{array}$$

The key step of this proof is the elimination of the universally quantified semantic type Ψ—which again demonstrates why "logical relations" are called "logical".

To prove the actual semantic typing rule S-tapp, we unfold the definition of the semantic typing judgment $\Delta;\Gamma \vDash e\langle\rangle : A[B/\alpha]$, which shows we have to prove that:

$$\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket A[B/\alpha] \rrbracket^{\mathrm{e}}_{\delta}(e[\vec{v}/\vec{x}]\langle\rangle)$$

follows from the assumption $\Delta;\Gamma \vDash e : \forall\alpha.\, A$, which in turn unfolds to:

$$\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v}) \mathrel{-\!\!*} \llbracket \forall\alpha.\, A \rrbracket^{\mathrm{e}}_{\delta}(e[\vec{v}/\vec{x}]).$$

This result follows by threading through $\llbracket \Gamma \rrbracket^{\mathrm{c}}_{\delta}(\vec{v})$, Lemma 6.2, and the auxiliary result for closed expressions that we proved above.

Proof of S-tlam. Following the usual approach, to prove the semantic typing rule, we first prove an auxiliary result for closed expressions:

$$\square\big(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_{\square}).\, \llbracket A \rrbracket^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)\big) \mathrel{-\!\!*} \llbracket \forall\alpha.\, A \rrbracket^{\mathrm{e}}_{\delta}(\Lambda.\, e)$$

This auxiliary result is proved as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![A]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e) \vdash \mathsf{wp}\ e\ \{[\![A]\!]_{\delta,\alpha\mapsto\Psi}\}}\ \text{unfold } [\![A]\!]^{\mathrm{e}} \text{ on LHS}}{[\![A]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e) \vdash \mathsf{wp}\ (\Lambda.\, e)\langle\rangle\ \{[\![A]\!]_{\delta,\alpha\mapsto\Psi}\}}\ \textsc{wp-pure}}{\square\,(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_\square).\, [\![A]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)) \vdash \square\,(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_\square).\, \mathsf{wp}\ (\Lambda.\, e)\langle\rangle\ \{[\![A]\!]_{\delta,\alpha\mapsto\Psi}\})}\ \square\textsc{-mono},\ \forall\text{-mono}}{\square\,(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_\square).\, [\![A]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)) \vdash [\![\forall\alpha.\, A]\!]_\delta(\Lambda.\, e)}\ \begin{array}{l}\text{unfold } [\![\forall\alpha.\, A]\!] \\ \text{and } [\![A]\!]^{\mathrm{e}} \text{ on RHS}\end{array}}{\square\,(\forall(\Psi : \mathit{Val} \to \mathit{iProp}_\square).\, [\![A]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)) \vdash [\![\forall\alpha.\, A]\!]^{\mathrm{e}}_\delta(\Lambda.\, e)}\ \textsc{logrel-val}$$

6.7 Recursive Types

Recall from Figure 5 the value interpretation for recursive types:

$$[\![\mu\alpha.\, A]\!]_\delta \triangleq \mu\,(\Psi : \mathit{Val} \to \mathit{iProp}_\square).\, \lambda v.\, \exists w.\, (v = \mathtt{fold}\, w) * \triangleright[\![A]\!]_{\delta,\alpha\mapsto\Psi}(w)$$

As explained in §5.1, the interpretation of recursive types ($\mu\alpha.\, A$) makes use of Iris's *guarded fixed-point* operator ($\mu\, x.\, t$), which can be used to define recursive predicates without a restriction on the variance of the recursive occurrences of x in t. Instead, all recursive occurrences of x must be *guarded, i.e.,* they have to appear below a later modality ($\triangleright$). In the above definition this means that $[\![A]\!]_{\delta,\alpha\mapsto\Psi}(w)$ must appear below a later. The later makes a proposition weaker—we have $P \vdash \triangleright P$, see $\triangleright$-INTRO, but not the inverse, as that would make the logic inconsistent. However, having $[\![A]\!]_{\delta,\alpha\mapsto\Psi}(w)$ below a later is strong enough for proving the semantic typing rules:

$$\textsc{S-fold}\quad \dfrac{\Delta; \Gamma \vDash e : A[\mu\alpha.\, A/\alpha]}{\Delta; \Gamma \vDash \mathtt{fold}\, e : \mu\alpha.\, A} \qquad\qquad \textsc{S-unfold}\quad \dfrac{\Delta; \Gamma \vDash e : \mu\alpha.\, A}{\Delta; \Gamma \vDash \mathtt{unfold}\, e : A[\mu\alpha.\, A/\alpha]}$$

As we will see, the proof of S-FOLD makes use of rule $\triangleright$-INTRO, *i.e.,* $P \vdash \triangleright P$, while the proof of S-UNFOLD crucially relies on the fact that a computation step is performed to strip off the later.

The proofs of the semantic typing rules make use of the following unfolding lemma.

LEMMA 6.3. $[\![\mu\alpha.\, A]\!]_\delta(v) = \big(\exists w.\, (v = \mathtt{fold}\, w) * \triangleright[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(w)\big)$.

PROOF. By μ-UNFOLD we obtain $[\![\mu\alpha.\, A]\!]_\delta(v) = \big(\exists w.\, (v = \mathtt{fold}\, w) * \triangleright[\![A]\!]_{\delta,\alpha\mapsto[\![\mu\alpha.\, A]\!]_\delta}(w)\big)$, which in turn by Lemma 6.2 concludes the proof. □

Proof of S-FOLD. To prove this rule, we first prove the following auxiliary result for closed expressions, from which the semantic typing rule follows immediately:

$$[\![A[\mu\alpha.\, A/\alpha]]\!]^{\mathrm{e}}_\delta(e) \mathrel{-\!\!*} [\![\mu\alpha.\, A]\!]^{\mathrm{e}}_\delta(\mathtt{fold}\, e)$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(v) \vdash \triangleright[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(v)}\ \triangleright\textsc{-intro}}{[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(v) \vdash \exists w.\, (v = \mathtt{fold}\, w) * \triangleright[\![A]\!]_{\delta,\alpha\mapsto[\![\mu\alpha.\, A]\!]_\delta}(w)}\ \exists\text{-intro}, =\text{-refl}}{[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(v) \vdash [\![\mu\alpha.\, A]\!]_\delta(\mathtt{fold}\, v)}\ \text{Lemma 6.3 on RHS}}{[\![A[\mu\alpha.\, A/\alpha]]\!]_\delta(v) \vdash [\![\mu\alpha.\, A]\!]^{\mathrm{e}}_\delta(\mathtt{fold}\, v)}\ \textsc{logrel-val}}{[\![A[\mu\alpha.\, A/\alpha]]\!]^{\mathrm{e}}_\delta(e) \vdash [\![\mu\alpha.\, A]\!]^{\mathrm{e}}_\delta(\mathtt{fold}\, e)}\ \textsc{logrel-bind}$$

Proof of S-UNFOLD. To prove this rule, we first prove the following auxiliary result for closed expressions, from which the semantic typing rule follows immediately:

$$[\![\mu\alpha.\, A]\!]^{\mathrm{e}}_\delta(e) \mathrel{-\!\!*} [\![A[\mu\alpha.\, A/\alpha]]\!]^{\mathrm{e}}_\delta(\mathtt{unfold}\, e)$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta(w) \vdash \mathsf{wp}\; w\,\{[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta\}}\;\textsc{wp-val}}{\triangleright[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta(w) \vdash \triangleright\,\mathsf{wp}\; w\,\{[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta\}}\;\triangleright\textsc{-mono}}{\triangleright[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta(w) \vdash \mathsf{wp}\; \mathtt{unfold}\,(\mathtt{fold}\,w)\,\{[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta\}}\;\textsc{wp-pure}}{\triangleright[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta(w) \vdash [\![A[\mu\alpha.\,A/\alpha]]\!]^{\mathsf{e}}_\delta(\mathtt{unfold}\,(\mathtt{fold}\,w))}\;\text{unfold } [\![A[\mu\alpha.\,A/\alpha]]\!]^{\mathsf{e}} \text{ on RHS}}{\exists w.\,(v = \mathtt{fold}\,w) * \triangleright[\![A]\!]_{\delta,\alpha\mapsto[\![\mu\alpha.\,A]\!]_\delta}(w) \vdash [\![A[\mu\alpha.\,A/\alpha]]\!]^{\mathsf{e}}_\delta(\mathtt{unfold}\,v)}\;\exists\text{-elim, =-subst}}{[\![\mu\alpha.\,A]\!]^{\mathsf{e}}_\delta(v) \vdash [\![A[\mu\alpha.\,A/\alpha]]\!]^{\mathsf{e}}_\delta(\mathtt{unfold}\,v)}\;\text{Lemma 6.3 on LHS}}{[\![\mu\alpha.\,A]\!]^{\mathsf{e}}_\delta(e) \vdash [\![A[\mu\alpha.\,A/\alpha]]\!]^{\mathsf{e}}_\delta(\mathtt{unfold}\,e)}\;\textsc{logrel-bind}$$

The key step of this proof is the use of rule wp-pure, whose premise contains a later, and thus allows stripping-off the later of the hypothesis $\triangleright[\![A[\mu\alpha.\,A/\alpha]]\!]_\delta(w)$ using $\triangleright$-mono.

It is worth noting that neither the proofs in this section, nor the proofs of any other semantic typing rule, involve explicit reasoning about step-indices. This kind of reasoning is encapsulated by Iris and becomes apparent only by a few judicious uses of the later modality ($\triangleright$).

6.8 Reference Types

Recall from Figure 5 the value interpretation for reference types:

$$[\![\mathtt{ref}(A)]\!]_\delta \triangleq \lambda\, v.\, \exists(\ell : Loc).\,(v = \ell) * \boxed{\exists w.\,\ell \mapsto w * [\![A]\!]_\delta(w)}^{\mathcal{N}_{\mathsf{ty}}.\ell}$$

As explained in §5.1, values of the reference type $\mathtt{ref}(A)$ should be memory locations ℓ at which the value w stored may change over time but is always of type A. This definition makes use of the *points-to connective* $\ell \mapsto v$ (from vanilla separation logic), which asserts exclusive ownership of the location ℓ storing value v, and Iris's *invariant assertion* $\boxed{P}^{\mathcal{N}}$, which expresses that a proposition P holds *invariantly—i.e.,* at all times. As explained in §6.1, $\ell \mapsto v$ asserts exclusive ownership and is thus an ephemeral (non-persistent) proposition. By wrapping it in in an invariant, we obtain a persistent proposition (which is thus freely duplicable).

The formal rules for invariants in Iris (inv-alloc, inv-persist, and inv-open-wp) can be found in Figure 7. Before we go into detail about these rules, let us informally explain the high-level roadmap for how one reasons about invariants in Iris:

(1) **Invariant allocation:** At any moment in an Iris proof, if one can assert ownership of a proposition P, one can give this up in exchange for creating an invariant $\boxed{P}^{\mathcal{N}}$ (the invariant namespace $\mathcal{N}$ can be ignored for now). This can be understood as a form of *ownership transfer*: P is being transferred from one's *private state* (*i.e.,* the private state of the thread whose code one is verifying) to the *shared state* (*i.e.,* state shared by all threads). This ownership transfer to obtain an invariant is called *allocating* an invariant.

(2) **Invariant duplication:** The upside of creating an invariant is that it enables one to take an ephemeral proposition (describing exclusive ownership of some state) and make it accessible to multiple threads at the same time. As explained above, this is achieved by the fact that the invariant assertion $\boxed{P}^{\mathcal{N}}$ is persistent: after an invariant has been allocated, it can be freely duplicated and thus shared among multiple threads.

(3) **Invariant access:** The downside of turning P into an invariant is that no thread has unfettered access to P anymore because it has become a shared resource. Rather, each thread may only access the resource governed by the invariant in a carefully restricted way: during any *atomic* step of computation, a thread may acquire exclusive ownership of P so long as it gives P back by the end of that step. Atomicity of invariant access is essential for soundness of invariants because in between acquiring and releasing ownership of P, the thread *does* have exclusive ownership, so it may in fact temporarily break the invariant (by falsifying P). But since this temporary breaking of the invariant only occurs *within* the reasoning about an atomic step

of computation, no other threads can observe it, so it does not cause any problems. We refer to the acquisition and release of the ownership of the contents of an invariant as the *opening* and *closing* of the invariant.

Opening and closing invariants. Iris's rule for opening invariants is INV-OPEN-WP:[19]

$$\textsc{inv-open-wp}\quad \frac{\mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \Big(\triangleright P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\ \triangleright P * \Phi(v)\}\Big) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

This rule is quite a mouthful, so let us go over it piece by piece. When proving a weakest precondition of an atomic expression e, this rule allows one to temporarily acquire exclusive ownership of P for the duration of the atomic step. Using the magic wand, one acquires $\triangleright P$ as an additional resource that can be used for proving the weakest precondition. In turn, in the postcondition of the weakest precondition, one has to restore $\triangleright P$. The side-condition $\mathsf{atomic}(e)$ makes sure the rule is only used for *physically atomic expressions*, *i.e.,* expressions e that take at most one step of computation. Examples of physically atomic expressions are $\mathtt{ref}(v)$, $!\,\ell$, $\ell \leftarrow v$, $\mathtt{FAA}(\ell, v)$, and $\mathtt{CAS}(\ell, v_1, v_2)$.

The reader may rightly wonder about the appearance of the $\triangleright$ modality in this rule. It turns out this is crucial for ensuring soundness in the presence of impredicative invariants. By *impredicativity* of invariants, we mean that the proposition P in $\boxed{P}^{\mathcal{N}}$ can be *any* Iris proposition, including one that contains nested invariant assertions. Impredicativity in turn is crucial for modeling the combination of polymorphism and higher-order references: the interpretation of a type like $\forall\alpha \ldots \mathtt{ref}(A) \ldots$ will quantify (universally) over an arbitrary predicate Ψ representing α, and then Ψ will appear inside the invariant modeling the reference type $\mathtt{ref}(A)$. As shown by Krebbers et al. [2017a] and Jung et al. [2018b], the $\triangleright$ modality helps to ensure that impredicative invariants do not introduce a subtle form of inconsistent circular reasoning into the logic.[20] Note, however, that after opening an invariant, one can use the step-taking weakest precondition rules (like WP-PURE, WP-ALLOC, WP-LOAD) in order to strip the $\triangleright$ modality off the assumed proposition $\triangleright P$, thus obtaining P for use "now" in proving the postcondition $\Phi(v)$. We will see an example of this in the proof of S-LOAD below.

Invariant namespaces and masks. Two other important Iris mechanisms—albeit largely administrative ones that serve to ensure soundness of invariant reasoning—are *invariant namespaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ where $\mathit{InvName} \triangleq \mathit{List}(\mathit{String} + \mathit{Val})$. Namespaces and masks are used to ensure that invariants cannot be opened twice in a nested fashion, *i.e.,* that a thread cannot acquire exclusive ownership of the contents of the same invariant twice during the same atomic step of computation—an issue often referred to as *reentrancy.* To avoid reentrancy, Iris annotates each invariant $\boxed{P}^{\mathcal{N}}$ with a *namespace* $\mathcal{N}$ that identifies the invariant, and annotates weakest preconditions $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ with a *mask* $\mathcal{E}$ that keeps track of the invariants that may be opened. At the top level, we always consider weakest preconditions with the mask $\top$, meaning that all invariants are available to be opened. Opening an invariant $\boxed{P}^{\mathcal{N}}$ removes the namespace $\mathcal{N}$ from the mask $\mathcal{E}$, ensuring that it cannot be opened in a nested fashion. Namespaces are simply (non-empty) lists of strings and values. For example, to identify the invariant for each reference ℓ, we use the namespace $\mathcal{N}_{\mathtt{ty}}.\ell$, where $\mathcal{N}_{\mathtt{ty}}$ is some fixed parent namespace chosen up front. As we

[19] The other rule for opening invariants, INV-OPEN-UPD, will be discussed in §7.

[20] The two occurrences of the $\triangleright$ modality in INV-OPEN-WP are not strictly needed for ensuring consistency. In particular, recent work by Spies et al. [2022] on "later credits" has shown how the first occurrence of $\triangleright$ in INV-OPEN-WP can be removed at the expensive of complicating the invariant allocation rule. And as for the second occurrence, it simply makes the rule stronger by weakening the postcondition that must be proved for e.

have implicitly done in the invariant opening rule INV-OPEN-WP, namespaces $\mathcal{N}$ are coerced into masks by taking the set of all extensions $\{\mathcal{N}.x_1.\cdots.x_n \mid n \in \mathbb{N}, x_i \in \mathit{String} + \mathit{Val}\}$.

Allocation of invariants. Using the following rules, one can transfer exclusive ownership of $\triangleright P$ into an invariant $\boxed{P}^{\mathcal{N}}$:

$$\triangleright P * \left(\boxed{P}^{\mathcal{N}} -\!\!* \; \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\; e\; \{\Phi\} \qquad (\textsc{inv-alloc-wp})$$

Iris in fact provides a more flexible rule for invariant allocation, called INV-ALLOC, from which the above rule is derived. We will discuss this more flexible rule in §7.

Semantic typing rules. Some of the semantic typing rules for references are:

$$\textsc{S-alloc}\;\dfrac{\Delta;\Gamma \vDash e : A}{\Delta;\Gamma \vDash \mathtt{ref}(e) : \mathtt{ref}(A)} \qquad \textsc{S-load}\;\dfrac{\Delta;\Gamma \vDash e : \mathtt{ref}(A)}{\Delta;\Gamma \vDash\, !\, e : A} \qquad \textsc{S-store}\;\dfrac{\Delta;\Gamma \vDash e_1 : \mathtt{ref}(A) \qquad \Delta;\Gamma \vdash e_2 : A}{\Delta;\Gamma \vDash e_1 \leftarrow e_2 : \mathbf{1}}$$

Proof of S-ALLOC. As usual, most of the work is in proving an auxiliary result for closed expressions. This auxiliary result for S-ALLOC is as follows:

$$[\![A]\!]^{\mathrm{e}}_{\delta}(e) -\!\!* \; [\![\mathtt{ref}(A)]\!]_{\delta}(\mathtt{ref}(e))$$

The following is a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ell \mapsto v * [\![A]\!]_\delta(v) \vdash \triangleright I_\ell}\;\triangleright\text{-INTRO, unfold } I_\ell, \exists\text{-intro} \qquad \dfrac{\dfrac{\dfrac{}{\boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell} \vdash \exists \ell'.\, (\ell = \ell') * \boxed{I_{\ell'}}^{\mathcal{N}_{\mathtt{ty}}.\ell'}}\;\exists\text{-intro, =-refl}}{\boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell} \vdash [\![\mathtt{ref}(A)]\!]_\delta(\ell)}\;\text{unfold } [\![\mathtt{ref}(A)]\!] \text{ on RHS}}{\boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell} \vdash \mathsf{wp}\; \ell\; \{[\![\mathtt{ref}(A)]\!]_\delta\}}\;\text{WP-VAL}}{\ell \mapsto v * [\![A]\!]_\delta(v) \vdash \triangleright I_\ell * \left(\boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell} -\!\!* \; \mathsf{wp}\; \ell\; \{[\![\mathtt{ref}(A)]\!]_\delta\}\right)}\;*\text{-MONO}}{\ell \mapsto v * [\![A]\!]_\delta(v) \vdash \mathsf{wp}\; \ell\; \{[\![\mathtt{ref}(A)]\!]_\delta\}}\;\text{INV-ALLOC-WP}}{[\![A]\!]_\delta(v) \vdash \forall \ell.\, \ell \mapsto v -\!\!* \; \mathsf{wp}\; \ell\; \{[\![\mathtt{ref}(A)]\!]_\delta\}}\;\forall\text{-intro, } -\!\!*\text{-INTRO}}{[\![A]\!]_\delta(v) \vdash \mathsf{wp}\; \mathtt{ref}(v)\; \{[\![\mathtt{ref}(A)]\!]_\delta\}}\;\text{WP-ALLOC}}{[\![A]\!]_\delta(v) \vdash [\![\mathtt{ref}(A)]\!]^{\mathrm{e}}_\delta(\mathtt{ref}(v))}\;\text{unfold } [\![\mathtt{ref}(A)]\!]^{\mathrm{e}} \text{ on RHS}}{[\![A]\!]^{\mathrm{e}}_\delta(e) \vdash [\![\mathtt{ref}(A)]\!]^{\mathrm{e}}_\delta(\mathtt{ref}(e))}\;\text{LOGREL-BIND}$$

Here, we let $I_\ell \triangleq \exists w.\, \ell \mapsto w * [\![A]\!]_\delta(w)$. The most important part of this proof is the use of the invariant allocation rule INV-ALLOC-WP. This rule allows us to transfer the exclusive ownership of $\ell \mapsto v$ and the interpretation $[\![A]\!]_\delta(v)$ into the invariant $\boxed{\exists w.\, \ell \mapsto w * [\![A]\!]_\delta(w)}^{\mathcal{N}_{\mathtt{ty}}.\ell}$ (*i.e.*, I_ℓ), which we in turn use to prove the value interpretation of $[\![\mathtt{ref}(A)]\!]_\delta(\ell)$.

Proof of S-LOAD. This rule follows immediately from the following auxiliary result for closed expressions:

$$[\![\mathtt{ref}(A)]\!]^{\mathrm{e}}_\delta(e) -\!\!* \; [\![A]\!]^{\mathrm{e}}_\delta(!\, e)$$

The following is a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ell \mapsto w * \llbracket A\rrbracket_\delta(w) \vdash \triangleright I_\ell}\;\triangleright\text{-}\textsc{intro}\text{, unfold } I_\ell\text{, } \exists\text{-intro} \qquad \dfrac{}{\llbracket A\rrbracket_\delta(w) \vdash \llbracket A\rrbracket_\delta(w)}}{\ell \mapsto w * \llbracket A\rrbracket_\delta(w) \vdash \triangleright I_\ell * \llbracket A\rrbracket_\delta(w)}\;{*}\text{-}\textsc{mono}\text{ and }\square\text{-}\textsc{dup}}{\ell \mapsto w * \llbracket A\rrbracket_\delta(w) \vdash \mathtt{wp}_{\top\setminus\mathcal{N}_{\mathtt{ty}}.\ell}\, w\, \{w.\ \triangleright I_\ell * \llbracket A\rrbracket_\delta(w)\}}\;\textsc{wp-val}}{\triangleright(\ell \mapsto w * \llbracket A\rrbracket_\delta(w)) \vdash \mathtt{wp}_{\top\setminus\mathcal{N}_{\mathtt{ty}}.\ell}\, !\,\ell\, \{w.\ \triangleright I_\ell * \llbracket A\rrbracket_\delta(w)\}}\;\textsc{wp-load}\text{, }\triangleright\text{-}\textsc{mono}}{\triangleright I_\ell \vdash \mathtt{wp}_{\top\setminus\mathcal{N}_{\mathtt{ty}}.\ell}\, !\,\ell\, \{w.\ \triangleright I_\ell * \llbracket A\rrbracket_\delta(w)\}}\;\text{unfold } I_\ell\text{, }\triangleright\text{-}\textsc{exists}\text{, }\exists\text{-elim on LHS}}{\boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell} \vdash \mathtt{wp}\ !\,\ell\, \{\llbracket A\rrbracket_\delta\}}\;\textsc{inv-open-wp}}{(\exists \ell.\ (v = \ell) * \boxed{I_\ell}^{\mathcal{N}_{\mathtt{ty}}.\ell}) \vdash \mathtt{wp}\ !\,v\, \{\llbracket A\rrbracket_\delta\}}\;\exists\text{-elim, =-subst}}{\llbracket \mathtt{ref}(A)\rrbracket_\delta(v) \vdash \mathtt{wp}\ !\,v\, \{\llbracket A\rrbracket_\delta\}}\;\text{unfold } \llbracket \mathtt{ref}(A)\rrbracket \text{ on LHS}}{\llbracket \mathtt{ref}(A)\rrbracket_\delta(v) \vdash \llbracket A\rrbracket^{\mathrm{e}}_\delta(!\,v)}\;\text{unfold } \llbracket A\rrbracket^{\mathrm{e}} \text{ on RHS}}{\llbracket \mathtt{ref}(A)\rrbracket^{\mathrm{e}}_\delta(e) \vdash \llbracket A\rrbracket^{\mathrm{e}}_\delta(!\,e)}\;\textsc{logrel-bind}$$

The most important part of this proof is the use of the invariant opening rule INV-OPEN-WP, using which we obtain temporary ownership of $\exists w.\ \ell \mapsto w * \llbracket A\rrbracket_\delta(w)$, as needed to prove the weakest precondition for the load operation. Notice that by using INV-OPEN-WP we get the resources $\triangleright(\exists w.\ \ell \mapsto w * \llbracket A\rrbracket_\delta(w))$ (*i.e.*, $\triangleright I_\ell$) with a later modality. However, since the load instruction takes a step of computation, after having used the load rule WP-LOAD (which contains a later modality in the premise), we can use the rule ▷-MONO to remove the later from our hypotheses, thus obtaining the resource $\ell \mapsto w * \llbracket A\rrbracket_\delta(w)$ "now". Finally, note that at the top of the proof we are free to duplicate $\llbracket A\rrbracket_\delta(w)$ because the value interpretation of types is persistent by construction.

The proof of S-STORE is similar to the proof of S-LOAD. The key part of the proof lies in the fact that the value w is existentially quantified in $\boxed{\exists w.\ \ell \mapsto w * \llbracket A\rrbracket_\delta(w)}^{\mathcal{N}_{\mathtt{ty}}.\ell}$. This means that it is fine to update ℓ to a new w so long as it satisfies $\llbracket A\rrbracket_\delta$ (as the second premise of S-STORE guarantees).

6.9 The Fundamental Theorem and Adequacy

So far we have shown semantic typing rules corresponding to all syntactic typing rules, except for T-CAS, T-FAA and T-FORK. We leave the proofs of the semantic versions of those rules to the reader. With those in hand we obtain that syntactic typing implies semantic typing:

THEOREM 6.4 (FUNDAMENTAL THEOREM OF UNARY LOGICAL RELATIONS). *Every syntactically well-typed term is semantically well-typed. Formally, if* $(\Delta; \Gamma \vdash e : A)$, *then* $(\Delta; \Gamma \vDash e : A)$.

PROOF. This theorem is proven by induction on the type derivation $(\Delta; \Gamma \vdash e : A)$. For each case in the induction proof, we use the corresponding semantic typing rule. □

THEOREM 6.5 (ADEQUACY OF UNARY LOGICAL RELATIONS). *Every closed semantically well-typed expression e is safe: If* $(\emptyset; \emptyset \vDash e : A)$, *then* $\mathrm{safe}(e)$.

PROOF. From $(\emptyset; \emptyset \vDash e : A)$, we obtain $\llbracket A\rrbracket^{\mathrm{e}}_\emptyset(e)$ by definition of the semantic typing relation, which in turn, by definition, is equivalent to a closed proof of $\mathtt{wp}\ e\ \{\llbracket A\rrbracket_\emptyset\}$. We now obtain $\mathrm{safe}(e)$ by Iris's adequacy theorem [Jung et al. 2018b; Krebbers et al. 2017a], which says that a closed proof of a weakest precondition implies safety. □

COROLLARY 6.6 (SEMANTIC TYPE SOUNDNESS). *Every closed syntactically well-typed expression e is safe. Formally, if* $(\emptyset; \emptyset \vdash e : A)$, *then* $\mathrm{safe}(e)$.

PROOF. Let us assume that we have $(\emptyset; \emptyset \vdash e : A)$. By the fundamental theorem (Theorem 6.4), we obtain $(\emptyset; \emptyset \vDash e : A)$. By adequacy (Theorem 6.5), we obtain $\mathrm{safe}(e)$, which concludes the proof. □

7 SAFE ENCAPSULATION OF UNSAFE FEATURES

In the previous section, we showed how the logical approach to type soundness in Iris can be used to establish the well-known type soundness theorem (Theorem 6.6): well-typed programs are safe. Of course, if all we wanted was to prove Theorem 6.6, logical/semantic type soundness would not be needed—syntactic type soundness would suffice. What the stronger logical/semantic type soundness affords us is the additional ability to ensure that our language provides proper support for *data abstraction*, and to exploit that data abstraction for modular reasoning.

Concretely, recall the “evil”, data abstraction-breaking `gremlin` operator from §3.1. It is easy to see that, although `gremlin` is syntactically safe, it *does not* satisfy logical/semantic type soundness. In particular, suppose we wanted to prove $\vDash \texttt{gremlin} : \mathbf{1}$. To do so, we would have to show a weakest-pre for `gremlin`, and the difficult case would be the one where `gremlin` nondeterministically updates some arbitrary memory cell ℓ to 0. Of course, in a separation logic like Iris, one cannot simply modify a location ℓ that one does not own: it could be owned by another part of the program or governed by a shared invariant, and either way, updating it to 0 could break whatever invariant or ownership assertion is currently imposed on it. So with our Iris-based semantic typing judgment in hand, we can happily declare `gremlin` a *persona non grata* in our programming language.

This is great news, but even greater is the *positive* thing we obtain from the guaranteed absence of features like `gremlin`: namely, the ability to verify safety of abstractions that are implemented internally using unsafe (syntactically ill-typed) features. We will now demonstrate this additional power by verifying safety of the `symbol` ADT from §3.2.

Recall the implementation of the `symbol` ADT:

$$\texttt{symbol\_type} \triangleq \exists \alpha.\, (\mathbf{1} \to \alpha) \times (\alpha \to \mathbf{2})$$

$$\texttt{symbol} \triangleq \texttt{let}\ c = \texttt{ref}(0)\ \texttt{in}\ \texttt{pack}\left\langle \mathbf{Z}, \begin{pmatrix} \lambda\,().\, \texttt{FAA}(c, 1), \\ \lambda\, s.\, \texttt{assert}\ (s < {!}\,c) \end{pmatrix} \right\rangle : \texttt{symbol\_type}$$

As we already explained in §3, the implementation employs a private integer counter c, which is allocated when the expression defining `symbol` is evaluated. The counter c is used as a perpetual source of fresh symbols. When the `gensym` function (the first closure returned by `symbol`) is called, it uses the fetch-and-add (`FAA`) instruction to atomically increment the value of c and return the previous value. Thus, when called repeatedly, `gensym` will return 0, 1, 2, and so on.

The `check` function (the second closure returned by `symbol`) checks validity of its symbol argument by checking that it is less than the current value of the counter. For this, it uses **MyLang**’s unsafe `assert` operation; hence, `check` is only safe to execute (*i.e.*, will not get stuck) if $s < {!}\,c$ indeed evaluates to `true`. Due to **MyLang**’s support for data abstraction, $s < {!}\,c$ *does* always evaluate to `true` in all well-typed contexts. We will now formalize this informal argument by proving the following semantic typing judgment:

$$\vDash \texttt{symbol} : \texttt{symbol\_type}$$

When proving that the `symbol` ADT is semantically well-typed at an existential type—here, $\texttt{symbol\_type} = \exists \alpha.\, (\mathbf{1} \to \alpha) \times (\alpha \to \mathbf{2})$—the key step is to choose the right *semantic type* for modeling α, *i.e.*, an Iris predicate $\Psi : \mathit{Val} \to \mathit{iProp}_{\square}$ that describes the valid values of the abstract type α. When the functions of the ADT are given a value v of type α, they can *assume* that v satisfies Ψ, and when they return a value v of type α, they must *establish* that v satisfies Ψ.

The difficulty in the case of the `symbol` ADT is that the valid values of type α *change* over time. Intuitively, at any given point during the execution of a program containing `symbol`, the valid values of type α will be the symbols that have been generated *so far*—these are represented by the integers that are smaller than the *current* value of the private integer counter c used in the implementation of `symbol`. But how do we describe this intuitive property as a (persistent) Iris predicate $\Psi : \mathit{Val} \to \mathit{iProp}_{\square}$? It must be a *state-dependent* predicate, meaning that it grows dynamically to be satisfied by more and more values as the state of c increases over time. How can we even define such a thing in Iris?

Naively, one might think that the following definition of Ψ should do the trick:

$$\Psi(v) \triangleq \exists n : \mathbb{N}.\, (v < n) * c \mapsto n$$

This definition appears to say that v is a valid symbol if it is less than the current value n pointed to by c. The problem is that semantic types must be *persistent*, but this definition is *not* persistent. It asserts exclusive ownership of c, which persistent predicates may not do. Moreover, if a value v satisfies Ψ now, there is no guarantee that it will continue to do so even after c gets updated. Intuitively, to make Ψ persistent, we will need some way of ensuring that valid symbols *stay* valid, which means we will need some way of enforcing the invariant that the counter value pointed to by c only grows larger over time. Toward that end, we now introduce one more feature of Iris, which is in fact one of its most powerful and defining features: *user-defined ghost state.*

User-defined ghost state in Iris. Modern separation logics provide a variety of mechanisms for ownership of auxiliary state, often called *ghost state.* Some well-known examples include ghost variables [O'Hearn 2007], permissions [Bornat et al. 2005], protocols [Dinsdale-Young et al. 2010; Svendsen and Birkedal 2014], and history/prophecy assertions about the past/future trace of execution [Fu et al. 2010; Jung et al. 2020]. These mechanisms do not denote ownership of *physical* state (*e.g.,* a location in the heap); rather, they describe *logical* state—*i.e.,* state that is useful to track in proofs, but which is not directly manifested in the physical state of the program being verified.

In this section, we will show how to use Iris's support for ghost state to encode the logical state of the counter in the `symbol` ADT, along with the property that it only grows larger over time, so that we can formulate an appropriate *persistent* predicate Ψ with which to model the ADT's abstract type α. To be as flexible as possible, Iris does not bake in a particular ghost state mechanism, but rather allows the user of the framework to "roll their own" form of *user-defined* ghost state. Rolling your own ghost state essentially involves choosing an appropriate "resource algebra" to represent the ghost state mechanism you want; once the resource algebra is chosen, the base proof rules of Iris allow you to derive a corresponding *ghost theory—i.e.,* a set of abstract predicates describing ownership of ghost state, together with axioms for manipulating them—on top of it.

The details of resource algebras, and how they can be used to derive ghost theories, are beyond the scope of this paper. Here, we will focus our attention on a handful of concrete instances of user-defined ghost theories, and refer the reader to Jung et al. [2018b] for more details about how such theories can be derived from suitably chosen resource algebras within Iris.

We begin by presenting a ghost theory that is directly relevant to the proof of the `symbol` ADT—namely, a theory of *ghost counters.* The connectives for ghost counters are as follows:

$$\hookrightarrow_{\bullet} : \mathit{GName} \to \mathbb{N} \to \mathit{iProp} \qquad \hookrightarrow_{>} : \mathit{GName} \to \mathbb{N} \to \mathit{iProp}_{\square}$$

Similar to locations $\ell \mapsto v$ in physical state, ghost counters $\gamma \hookrightarrow_{\bullet} m$ and $\gamma \hookrightarrow_{>} n$ can be referred to by a name $\gamma \in \mathit{GName}$. Ghost counters can be allocated at any time during a proof and come in pairs: $\gamma \hookrightarrow_{\bullet} n$ is an ephemeral proposition that provides exclusive ownership of the ghost location and says its value is *exactly* n, while $\gamma \hookrightarrow_{>} m$ is a persistent proposition that says its value is *strictly greater than* m. Since $\gamma \hookrightarrow_{>} m$ provides *persistent* knowledge, the value of the ghost location γ can only ever be increased—decreasing it could result in an already-established persistent assertion $\gamma \hookrightarrow_{>} m$ becoming falsified, which is not something that Iris lets happen to persistent assertions.

$$\mathsf{True} \vdash |\!\!\Rrightarrow_{\mathcal{E}} \exists \gamma.\, \gamma \hookrightarrow_{\bullet} 0 \qquad (\textsc{cnt-init})$$

$$\gamma \hookrightarrow_{\bullet} n \vdash |\!\!\Rrightarrow_{\mathcal{E}} \gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n \qquad (\textsc{cnt-inc})$$

$$\gamma \hookrightarrow_{>} m \vdash \Box(\gamma \hookrightarrow_{>} m) \qquad (\textsc{cnt-persist})$$

$$\gamma \hookrightarrow_{\bullet} n * \gamma \hookrightarrow_{>} m \vdash m < n \qquad (\textsc{cnt-lt})$$

$$\mathsf{timeless}(\gamma \hookrightarrow_{\bullet} n) \text{ and } \mathsf{timeless}(\gamma \hookrightarrow_{>} n) \qquad (\textsc{cnt-timeless})$$

Fig. 8. Iris's rules for ghost counters.

Ghost counters are used in the proof of semantic typing of the `symbol` ADT as follows:

$$I \triangleq \boxed{\exists n : \mathbb{N}.\, c \mapsto n * \gamma \hookrightarrow_{\bullet} n}^{\mathcal{N}_{\mathsf{sym}}}$$

$$\Psi(v) \triangleq \exists m : \mathbb{N}.\, (v = m) * \gamma \hookrightarrow_{>} m$$

The invariant I, which will be shared by the closures of the ADT, describes that the value of the physical location c matches up with the value of the ghost counter at all times. The predicate Ψ, which is used for the interpretation of the abstract type α, employs the persistent part of the ghost counter $\gamma \hookrightarrow_{>} m$ to ensure that the values of type α are integers m that are smaller than the value stored in the counter c.

The rules for ghost counters are given in Figure 8. These rules make use of a new connective called the *update modality* $|\!\!\Rrightarrow_{\mathcal{E}}$. For now, think of the update modality as a connective that signifies modifications to ghost state. Let us go over the rules one by one:

- The rule cnt-init is used to allocate a new ghost counter. It provides exclusive ownership of $\gamma \hookrightarrow_{\bullet} 0$, where γ is a fresh (*i.e.,* existentially quantified) name.
- The rule cnt-inc is used to increment the ghost counter. In addition to transforming $\gamma \hookrightarrow_{\bullet} n$ into $\gamma \hookrightarrow_{\bullet} (n+1)$, the rule also yields $\gamma \hookrightarrow_{>} n$, which provides the persistent knowledge that the ghost counter is strictly greater than n. (Note: $*$ binds more tightly than $|\!\!\Rrightarrow_{\mathcal{E}}$, so "$|\!\!\Rrightarrow_{\mathcal{E}} P * Q$" means "$|\!\!\Rrightarrow_{\mathcal{E}} (P * Q)$".)
- The rule cnt-persist states that the connective $\gamma \hookrightarrow_{>} m$ is indeed persistent.
- The rule cnt-lt states that if we have exclusive ownership of γ (with current value n), along with the knowledge that γ's value is greater than m, then we must know that $m < n$.

The update modality. The update modality $|\!\!\Rrightarrow_{\mathcal{E}} Q$ has many similarities with the weakest precondition connective $\mathsf{wp}_{\mathcal{E}}\, e\, \{w.\, Q\}$, but is used for reasoning about ghost state rather than physical state. Since ghost state is merely logical, there is no physical program e, and the postcondition is merely a proposition, not a value predicate (*i.e.,* it does not have a binder w for the return value). The update modality can be used for the following purposes:

- In order for clients of a ghost theory to make use of rules for allocating or updating ghost state (like cnt-init and cnt-inc in Figure 8), Iris provides the rules $|\!\!\Rrightarrow$-wp and wp-$|\!\!\Rrightarrow$. These rules allow one to eliminate update modalities around weakest preconditions:

  $|\!\!\Rrightarrow$-wp
  $$|\!\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

  wp-$|\!\!\Rrightarrow$
  $$\mathsf{wp}_{\mathcal{E}}\, e\, \big\{w.\, |\!\!\Rrightarrow_{\mathcal{E}} \Phi(w)\big\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

  (In a more traditional presentation with Hoare triples, these rules would correspond to a strengthened rule of consequence, in which the implications for adjusting the pre- and postcondition of the Hoare triple were additionally permitted to perform ghost updates.) Apart from these rules, there are a number of administrative rules, shown in Figure 7. The update modality is monotone ($|\!\!\Rrightarrow$-mono), propositions can be moved under the update modality ($|\!\!\Rrightarrow$-intro), the update modality is transitive ($|\!\!\Rrightarrow$-trans), and propositions can be framed below the update modality ($|\!\!\Rrightarrow$-frame). Using these administrative rules, we can turn the rule cnt-inc into the following, more usable rule:

CNT-INC'
$$\dfrac{\gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}{\gamma \hookrightarrow_{\bullet} n * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

A proof tree for this rule is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}{|\!\!\Rrightarrow_{\mathcal{E}} \gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n * Q \vdash |\!\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\ |\!\!\Rrightarrow\text{-MONO}}{(|\!\!\Rrightarrow_{\mathcal{E}} \gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n) * Q \vdash |\!\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\ |\!\!\Rrightarrow\text{-FRAME}}{\gamma \hookrightarrow_{\bullet} n * Q \vdash |\!\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\ \text{CNT-INC}}{\gamma \hookrightarrow_{\bullet} n * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\ |\!\!\Rrightarrow\text{-WP}$$

This proof shows a typical proof pattern in Iris. We use a rule like cnt-inc in one of the hypotheses, and use $|\!\!\Rrightarrow$-wp to obtain a matching update modality in the goal. Then, using $|\!\!\Rrightarrow$-frame, we move all other hypotheses below the update modality, and finally use $|\!\!\Rrightarrow$-mono to introduce the update modality. Note that when mechanizing Iris proofs in Coq, the Iris Proof Mode [Krebbers et al. 2017b, 2018] takes care of these administrative steps automatically.

- In order to build modular logical abstractions (as we will show in §8.3), Iris's update modality can also be used for allocating and opening invariants. Similar to weakest preconditions, the update modality $|\!\!\Rrightarrow_{\mathcal{E}}$ is thus equipped with a mask $\mathcal{E}$ that denotes which invariants may be opened. The rules inv-alloc and inv-open-upd for opening invariants around the update modality are as follows:

INV-ALLOC
$$\triangleright P \vdash |\!\!\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-OPEN-UPD
$$\dfrac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} |\!\!\Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\triangleright P * Q)) \vdash |\!\!\Rrightarrow_{\mathcal{E}} Q}$$

The rule inv-alloc transfers ownership of a proposition P into an invariant $\boxed{P}^{\mathcal{N}}$. The rule inv-alloc-wp for allocating invariants around weakest preconditions can be derived from inv-alloc and $|\!\!\Rrightarrow$-wp. The rule inv-open-upd is very similar to the rule inv-open-wp that we have seen before, except with update modalities instead of weakest precondition assertions.

- When opening an invariant $\boxed{P}^{\mathcal{N}}$ via the rules inv-open-upd and inv-open-wp, one temporarily gets ownership of $\triangleright P$ where P is guarded by a later modality ($\triangleright$). As we have discussed in §6.8, the later modality is crucial for soundness in the presence of impredicative invariants; however, for the class of so-called *timeless* Iris propositions, one can acquire access to the contents of the invariant without a later:

INV-OPEN-UPD-TL
$$\dfrac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(P)}{\boxed{P}^{\mathcal{N}} * (P \mathrel{-\!\!*} |\!\!\Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (P * Q)) \vdash |\!\!\Rrightarrow_{\mathcal{E}} Q}$$

INV-OPEN-WP-TL
$$\dfrac{\mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(P)}{\boxed{P}^{\mathcal{N}} * \Big(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, P * \Phi(v)\}\Big) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

The formal definition of timelessness can be found in [Krebbers et al. 2017a; Jung et al. 2018b]; for the purpose of this paper, it is sufficient to think of timeless propositions as those whose meaning is independent of step-indexing (*i.e.*, is the same at every step-index). The class of timeless propositions is closed under the connectives of first-order logic (truth, falsum, conjunction, disjunction, implication, and universal and existential quantification), separation logic (separating conjunction, magic wand, and the points-to connective), the persistence modality, and Iris's connectives for ghost ownership (*e.g.*, ghost counters). Propositions that are *not* timeless include invariant and weakest precondition assertions, as well as the later and update modalities. In practice, it is common in Iris to establish invariants $\boxed{P}^{\mathcal{N}}$ where P is timeless and hence the above $\triangleright$-free rules apply.

Instead of baking in the rules INV-OPEN-UPD-TL and INV-OPEN-WP-TL as primitives, Iris provides the primitive rule $\mathrel{|\!\Rrightarrow}$-TIMELESS for removing a later of a timeless proposition:

$$\begin{array}{c}\mathrel{|\!\Rrightarrow}\text{-TIMELESS}\\ \dfrac{\text{timeless}(P)}{\triangleright P \vdash \mathrel{|\!\Rrightarrow}_{\mathcal{E}} P}\end{array}$$

The rules INV-OPEN-UPD-TL and INV-OPEN-WP-TL follow from the ordinary rules for opening invariants (INV-OPEN-UPD and INV-OPEN-WP, respectively) and $\mathrel{|\!\Rrightarrow}$-TIMELESS. Apart from brevity, the rule $\mathrel{|\!\Rrightarrow}$-TIMELESS also provides more flexibility. When dealing with invariants that contain both a timeless and a non-timeless part, one can remove the later of just the timeless part.

With the ghost counters and the rules about the update modality in hand, we are now ready to proceed with the proof of semantic typing of the symbol ADT.

The proof of the `symbol` ADT. To prove the semantic typing $\vDash \texttt{symbol} : \texttt{symbol\_type}$, we unfold the definition and see that we must first prove a result about the expression interpretation:

$$\llbracket \texttt{symbol\_type} \rrbracket^{\mathrm{e}}_{\delta}(\texttt{symbol})$$

The proof of this property is as follows.

$$\begin{array}{cl}
\dfrac{\text{proof of gensym}}{I_\gamma(\ell) \vdash \llbracket \mathbf{1} \rightarrow \alpha \rrbracket_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, ().\, \mathsf{FAA}(\ell, 1))} \quad \dfrac{\text{proof of check}}{I_\gamma(\ell) \vdash \llbracket \alpha \rightarrow \mathbf{2} \rrbracket_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, s.\, \mathsf{assert}(s < !\,\ell))} & \\ \hline
I_\gamma(\ell) \vdash \exists \Phi.\ \llbracket (\mathbf{1} \rightarrow \alpha) \times (\alpha \rightarrow \mathbf{2}) \rrbracket_{\delta, \alpha \mapsto \Phi} \begin{pmatrix} \lambda\, ().\, \mathsf{FAA}(\ell, 1), \\ \lambda\, s.\, \mathsf{assert}(s < !\,\ell) \end{pmatrix} & \exists\text{-intro} \\ \hline
I_\gamma(\ell) \vdash \llbracket \texttt{symbol\_type} \rrbracket_{\delta}\ (\mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle) & \text{unfold } \llbracket \texttt{symbol\_type} \rrbracket \\ \hline
I_\gamma(\ell) \vdash \mathsf{wp}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{WP-VAL} \\ \hline
\ell \mapsto 0 * \gamma \hookrightarrow_{\bullet} 0 \vdash \mathsf{wp}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{INV-ALLOC} \\ \hline
\ell \mapsto 0 \vdash \mathsf{wp}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{CNT-INIT} \\ \hline
\ell \mapsto 0 \vdash \mathsf{wp}\ \mathsf{let}\ c = \ell\ \mathsf{in}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{WP-PURE} \\ \hline
\ell \mapsto 0 \vdash \mathsf{wp}\ \ell\ \{v.\ \mathsf{wp}\ \mathsf{let}\ c = v\ \mathsf{in}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\}\} & \text{WP-VAL} \\ \hline
\vdash \mathsf{wp}\ \mathsf{ref}(0)\ \{v.\ \mathsf{wp}\ \mathsf{let}\ c = v\ \mathsf{in}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\}\} & \text{WP-ALLOC} \\ \hline
\vdash \mathsf{wp}\ \mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{pack}\, \langle \mathrm{Z}, \ldots \rangle\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{WP-BIND} \\ \hline
\vdash \mathsf{wp}\ \texttt{symbol}\ \{\llbracket \texttt{symbol\_type} \rrbracket_{\delta}\} & \text{unfold symbol} \\ \hline
\vdash \llbracket \texttt{symbol\_type} \rrbracket^{\mathrm{e}}_{\delta}(\texttt{symbol}) & \text{unfold } \llbracket \texttt{symbol\_type} \rrbracket^{\mathrm{e}}
\end{array}$$

Reading this proof tree bottom-up, as usual, we see that it comprises the following steps:

(1) Symbolically execute the expression `symbol`, thereby obtaining exclusive ownership of the private counter location ℓ with initial value 0.
(2) Use rule CNT-INIT to allocate ghost counter γ with initial value 0 (implicitly here, we use the same pattern as in the proof of CNT-INC' to eliminate the update modality).
(3) Transfer ownership of both into a new counter invariant $I_\gamma(\ell)$, defined as follows:

$$I_\gamma(\ell) \triangleq \boxed{\exists n : \mathbb{N}.\ c \mapsto n * \gamma \hookrightarrow_{\bullet} n}^{\mathcal{N}_{\mathrm{sym}}}$$

(4) The remaining goal is to prove that the value produced by executing `symbol` inhabits the value interpretation of `symbol_type` $\triangleq \exists \alpha.\, (\mathbf{1} \to \alpha) \times (\alpha \to \mathbf{2})$. Correspondingly, we choose as our interpretation of α the semantic type $\Psi_\gamma(v)$, defined as follows:

$$\Psi_\gamma(v) \triangleq \exists m : \mathbb{N}.\, (v = m) * \gamma \hookrightarrow_{>} m$$

(5) The proof then splits into the following two subgoals, corresponding to the `gensym` and `check` operations of the ADT.

$$I_\gamma(\ell) \vdash [\![\mathbf{1} \to \alpha]\!]_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, ().\, \mathtt{FAA}(\ell, 1))$$
$$I_\gamma(\ell) \vdash [\![\alpha \to \mathbf{2}]\!]_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, s.\, \mathtt{assert}(s < {!\,\ell}))$$

We now proceed to prove these auxiliary results.

The proof of `gensym`.

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \dfrac{
     \dfrac{
      \dfrac{
       \dfrac{
        \dfrac{}{\ell \mapsto (n+1) * \gamma \hookrightarrow_{\bullet} (n+1) \vdash (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m)}
        \quad
        \dfrac{\dfrac{}{\gamma \hookrightarrow_{>} n \vdash \exists k.\, (n = k) * \gamma \hookrightarrow_{>} k}}{\gamma \hookrightarrow_{>} n \vdash \Psi_\gamma(n)}\;\text{unfold } \Psi
       }{\ell \mapsto (n+1) * \gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n \vdash (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \Psi_\gamma(n)}\;\textsc{$*$-mono}
      }{\ell \mapsto (n+1) * \gamma \hookrightarrow_{\bullet} (n+1) * \gamma \hookrightarrow_{>} n \vdash \mathsf{wp}_{\top \setminus \mathcal{N}_{\mathsf{sym}}}\ n\ \{v.\, (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \Psi_\gamma(v)\}}\;\textsc{wp-val}
     }{\ell \mapsto (n+1) * \gamma \hookrightarrow_{\bullet} n \vdash \mathsf{wp}_{\top \setminus \mathcal{N}_{\mathsf{sym}}}\ n\ \{v.\, (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \Psi_\gamma(v)\}}\;\textsc{cnt-inc'}
    }{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n \vdash \mathsf{wp}_{\top \setminus \mathcal{N}_{\mathsf{sym}}}\ \mathtt{FAA}(\ell, 1)\ \{v.\, (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \Psi_\gamma(v)\}}\;\textsc{wp-faa}
   }{I_\gamma(\ell) \vdash \mathsf{wp}\ \mathtt{FAA}(\ell, 1)\ \{\Psi_\gamma\}}\;\textsc{inv-open-wp-tl}
  }{I_\gamma(\ell) \vdash \mathsf{wp}\ \ ((\lambda\, ().\, \mathtt{FAA}(\ell, 1))\ ())\ \ \{\Psi_\gamma\}}\;\textsc{wp-pure}
 }{I_\gamma(\ell) \vdash \Box\, \forall v.\, (v = ()) \twoheadrightarrow \mathsf{wp}\ \ ((\lambda\, ().\, \mathtt{FAA}(\ell, 1))\ v)\ \ \{\Psi_\gamma\}}\;\textsc{$\Box$-mono}, \textsc{$\twoheadrightarrow$-intro}, \text{subst } v
}{I_\gamma(\ell) \vdash [\![\mathbf{1} \to \alpha]\!]_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, ().\, \mathtt{FAA}(\ell, 1))}\;\text{unfold } [\![\mathbf{1} \to \alpha]\!]
$$

The beginning of this proof is like the proofs we have seen before: we unfold the expression interpretation, after which we have to prove a weakest precondition. To prove the weakest precondition for $\mathtt{FAA}(\ell, 1)$, we need to get temporary ownership of the points-to connective $\ell \mapsto n$, which we do by opening the invariant $I_\gamma(\ell)$. Since the invariant $I_\gamma(\ell)$ is timeless, we can use the rule INV-OPEN-WP-TL to acquire ownership of $I_\gamma(\ell)$ without the later modality. After we have used the rule WP-FAA, we use the rule CNT-INC to update the ghost counter $\gamma \hookrightarrow_{\bullet} n$ to $\gamma \hookrightarrow_{\bullet} (n+1)$, as needed to restore the invariant $I_\gamma(\ell)$. By using CNT-INC, we also get $\gamma \hookrightarrow_{>} n$, which we need to establish $\Psi_\gamma(n)$.

The proof of check.

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \dfrac{
     \dfrac{
      \dfrac{
       \dfrac{
        \dfrac{
         \dfrac{
          \dfrac{}{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n \vdash (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m)}
          \quad
          \dfrac{\dfrac{\dfrac{}{\gamma \hookrightarrow_{>} k * k < n \vdash \mathtt{true} \in \{\mathtt{true}, \mathtt{false}\}}}{\gamma \hookrightarrow_{>} k * k < n \vdash \mathsf{wp}\ \mathtt{true}\ \{[\![2]\!]\}}\;\textsc{wp-val}, \text{unfold } [\![2]\!]}{\gamma \hookrightarrow_{>} k * k < n \vdash \mathsf{wp}\ \mathtt{assert}(k < n)\ \{[\![2]\!]\}}\;\textsc{wp-pure}
         }{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n * \gamma \hookrightarrow_{>} k * k < n \vdash (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \mathsf{wp} \ldots}\;\textsc{$*$-mono}
        }{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n * \gamma \hookrightarrow_{>} k \vdash (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \mathsf{wp} \ldots}\;\textsc{cnt-lt} \text{ on LHS}
       }{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n * \gamma \hookrightarrow_{>} k \vdash \mathsf{wp}_{\top \setminus \mathcal{N}_{\mathsf{sym}}}\ n\ \{w.\, (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \mathsf{wp} \ldots\}}\;\textsc{wp-val}
      }{\ell \mapsto n * \gamma \hookrightarrow_{\bullet} n * \gamma \hookrightarrow_{>} k \vdash \mathsf{wp}_{\top \setminus \mathcal{N}_{\mathsf{sym}}}\ {!\,\ell}\ \{w.\, (\exists m.\, \ell \mapsto m * \gamma \hookrightarrow_{\bullet} m) * \mathsf{wp} \ldots\}}\;\textsc{wp-load}
     }{I_\gamma(\ell) * \gamma \hookrightarrow_{>} k \vdash \mathsf{wp}\ {!\,\ell}\ \{w.\, \mathsf{wp}\ \mathtt{assert}(k < w)\ \{[\![2]\!]\}\}}\;\textsc{inv-open-wp-tl}
    }{I_\gamma(\ell) * \gamma \hookrightarrow_{>} k \vdash \mathsf{wp}\ \mathtt{assert}(k < {!\,\ell})\ \{[\![2]\!]\}}\;\textsc{wp-bind}
   }{I_\gamma(\ell) * \gamma \hookrightarrow_{>} k \vdash \mathsf{wp}\ (\lambda\, s.\, \mathtt{assert}(s < {!\,\ell}))\ k\ \{[\![2]\!]\}}\;\textsc{wp-pure}
  }{I_\gamma(\ell) \vdash \forall v.\, \Psi_\gamma(v) \twoheadrightarrow \mathsf{wp}\ (\lambda\, s.\, \mathtt{assert}(s < {!\,\ell}))\ v\ \{[\![2]\!]\}}\;\twoheadrightarrow, \forall \text{ intro, unpack } \Psi
 }{I_\gamma(\ell) \vdash \Box\, \forall v.\, \Psi_\gamma(v) \twoheadrightarrow \mathsf{wp}\ (\lambda\, s.\, \mathtt{assert}(s < {!\,\ell}))\ v\ \{[\![2]\!]\}}\;\textsc{$\Box$-mono}
}{I_\gamma(\ell) \vdash [\![\alpha \to 2]\!]_{\delta, \alpha \mapsto \Psi_\gamma}\ (\lambda\, s.\, \mathtt{assert}(s < {!\,\ell}))}\;\text{unfold } [\![\alpha \to 2]\!]
$$

This proof is similar structurally to the proof of gensym—to prove the weakest precondition for $!\,\ell$, we need temporary access to the points-to connective $\ell \mapsto n$, which we do by opening the invariant $I_\gamma(\ell)$ using INV-OPEN-WP-TL. Apart from the points-to connective, the invariant $I_\gamma(\ell)$ also provides temporary access to $\gamma \hookrightarrow_{\bullet} n$, which, using CNT-LT, allows us that to conclude that $k < n$ and hence that assert$(k < n)$ is safe.

Concluding remarks. In this section, we have demonstrated how the logical relation for semantic soundness, encoded in Iris, can be used to reason about safe encapsulation of unsafe features. As shown in the RustBelt project [Jung et al. 2018a, 2021; Jung 2020; Dang et al. 2020], this approach scales up to much more complicated uses of unsafe features. Of course, when dealing with more complicated uses of unsafe features, one needs more complicated invariants and "ghost theories", but the basic structure of the proofs nevertheless follows the template we have shown here.

8 REPRESENTATION INDEPENDENCE

In the previous sections, we have shown how the semantic approach can be used to prove type safety. However, the semantic approach is by no means limited to type safety—it can be used for the verification of many program properties, including but not limited to compiler correctness [Benton and Hur 2009], capability safety, both for object capabilities [Devriese et al. 2016] and capability machines [Georges et al. 2021], non-interference [Frumin et al. 2021a; Gregersen et al. 2021], and contextual refinement and representation independence, the topic of this section. Many of these properties are not about the execution of a single program, *i.e.,* they are not *unary* properties, but are rather about the relation between multiple runs of possibly different programs, *i.e.,* they are *binary*, or n-ary, properties. In this section, we discuss how Iris can be used to apply the semantic approach to prove a particularly important binary program property—*contextual refinement*—and in particular, the instance of that property known as *representation independence*.

A program e is said to *contextually refine* a program e', written $\Delta; \Gamma \vDash e \leq_{ctx} e' : A$, if for all program contexts C (with hole of type A), if $C[e]$ has some (observable) behavior, then so does $C[e']$. Contextual refinement $e \leq_{ctx} e'$ is a strong notion—whenever e' appears as part of a well-typed program (*i.e.,* plugged into a well-typed context C), then e' can be replaced by e without changing the (observable) behavior of the program.

A particularly interesting application of contextual refinement is in relational reasoning about ADTs. Specifically, suppose we have two different ADTs, M_1 and M_2, which have the same interface, *i.e.,* the same type, but with different implementations, *e.g.,* different representations of the abstract type or of the internal state managed by it. Then if we can prove M_1 refines M_2, we know that any program that is written against the common interface of these ADTs can be linked with M_1 instead of M_2, and this should not have any observable effect, *despite* the fact that the ADTs are implemented differently. This property is known as *representation independence.* In practice, representation independence can be used to show that it is sound to replace a (possibly inefficient) reference implementation M_2 by an optimized implementation M_1. Correspondingly, in a contextual refinement $\Delta; \Gamma \vDash e \leq_{ctx} e' : A$, we often refer to e as the *implementation* and e' as the *specification.*

The definition of contextual refinement (see Definition 8.1) is expressed in terms of quantifying over *all* possible program contexts C. This quantification over all contexts C makes direct proofs of contextual refinement difficult in practice—carrying out a proof by induction on the context C is known to be tedious and complicated, and infeasible for even moderately small programs. In order to ease such proofs, we will define a judgment $\Delta; \Gamma \vDash e \leq_{log} e' : A$ for *logical refinement*, using *binary logical relations.* The high-level structure of the binary logical relations method is similar to the high-level structure of the unary method for semantic typing we have already seen.

- **Soundness.** We prove a *soundness* theorem, which states that the logical relation is sound with respect to contextual refinement:

  $$\Delta; \Gamma \vDash e \leq_{log} e' : A \text{ implies } \Delta; \Gamma \vDash e \leq_{ctx} e' : A.$$

  This result is similar to the adequacy theorem for semantic typing (which says that programs that are logically typed are safe).
- **Compatibility lemmas.** We show that the logical relation is compatible with syntactic typing. For instance, for function applications (the typing rule T-APP) in **MyLang** we show that the premises of the following compatibility lemma imply its conclusion:

  $$\frac{\Delta; \Gamma \vDash e_1 \leq_{log} e_1' : A \to B \qquad \Delta; \Gamma \vDash e_2 \leq_{log} e_2' : A}{\Delta; \Gamma \vDash e_1\ e_2 \leq_{log} e_1'\ e_2' : B}$$

  These compatibility lemmas are similar to the semantic typing rules.

These results can then be combined with manual proofs of logical refinements of ADTs to modularly prove contextual refinements of larger programs. For example, suppose we have manually proven $\emptyset; \emptyset \vDash e_1 \leq_{log} e_2 : A$, and suppose C is a (closed) context (of type B) with a hole of type A. It is an easy corollary of the above properties that we can obtain $\emptyset; \emptyset \vDash C[e_1] \leq_{ctx} C[e_2] : B$. (We will see a more general version of this corollary in §8.5.)

To define the binary logical relation for logical refinement, we follow the same pattern as we have used for the unary logical relation for semantic typing—with the main difference that we generalize all notions to pairs of values. That is, we define binary interpretations on pairs of closed values $[\![\_]\!]$, and pairs of closed expressions $[\![\_]\!]^{e}$, and then use these binary interpretations to define our logical relation for open programs, $\Delta; \Gamma \vDash e \leq_{log} e' : A$. The binary value interpretations are straightforward generalizations of their unary counterparts. For example, values of base type (unit, Boolean, and integer) are related if they are equal, and values of function type are related if, given related inputs, they have related results. The crux of the difference between the unary and binary logical relations is in the expression relation $[\![\tau]\!]^{e}(e, e')$, which expresses that e refines e'. To formalize this refinement in Iris, we use both weakest preconditions and Iris's ghost theory.

We proceed in this section with a formal definition of contextual refinement (§8.1). We then show how to generalize the value and expression interpretations to the binary case (§8.2 and 8.3, respectively). Subsequently, we prove the compatibility lemmas for the binary logical relation (§8.4), and prove that the logical relation is sound with respect to contextual refinement (§8.5). Finally, we demonstrate the ability to reason about representation independence of ADTs by proving that a fine-grained implementation of a concurrent stack refines a coarse-grained version (§8.6).

8.1 Contextual Refinement

To formally define the contextual refinement judgment $\Delta; \Gamma \vDash e \leq_{ctx} e' : A$, we first need to define the notion of program contexts. Figure 9 shows an excerpt of the grammar and the typing rules for well-typed contexts. We write $C : (\Delta \mid \Gamma; A) \rightsquigarrow (\Delta' \mid \Gamma'; A')$ to say that the context C is a program of type A' (closed under Δ' and Γ') with a hole that can be filled with any program of type A (closed under Δ and Γ). The typing rules for well-typed contexts in Figure 9 imply that whenever $\Delta; \Gamma \vdash e : A$ and $C : (\Delta \mid \Gamma; A) \rightsquigarrow (\Delta' \mid \Gamma'; A')$ hold, so does $\Delta'; \Gamma' \vdash C[e] : A'$, capturing the intuitive idea that well-typed contexts are just well-typed programs with a hole.

Definition 8.1 (Contextual refinement). We say *e contextually refines e'*, written $\Delta; \Gamma \vDash e \leq_{ctx} e' : A$, if $\Delta; \Gamma \vdash e : A$, and $\Delta; \Gamma \vdash e' : A$, and:

$$\forall C : (\Delta \mid \Gamma; A) \rightsquigarrow (\emptyset \mid \emptyset; \mathbf{1}).\, C[e] \downarrow \implies C[e'] \downarrow$$

$$\begin{aligned} C ::= {} & [\,] \mid \texttt{rec}\, f(x) = C \mid C\, e \mid e\, C \mid \Lambda.\, C \mid C\langle\rangle \mid C \circledcirc e \mid e \circledcirc C \mid \\ & \texttt{if}\, C\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{if}\, e\, \texttt{then}\, C\, \texttt{else}\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, C \mid \\ & \texttt{fold}\, C \mid \texttt{unfold}\, C \mid \texttt{ref}(C) \mid\, !\,C \mid C \leftarrow e \mid e \leftarrow C \mid \\ & \texttt{CAS}(C, e, e) \mid \texttt{CAS}(e, C, e) \mid \texttt{CAS}(e, e, C) \mid \texttt{FAA}(C, e) \mid \texttt{FAA}(e, C) \mid \texttt{fork}\ \{C\} \mid \ldots \end{aligned}$$

$$\textsc{C-rec}\quad \frac{C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma, x : A, f : A \to B; B)}{\texttt{rec}\, f(x) = C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; B)} \qquad \textsc{C-tlam}\quad \frac{C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta, \alpha \mid \Gamma; A)}{\Lambda.\, C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; \forall \alpha.\, A)}$$

$$\textsc{C-app}_1\quad \frac{\begin{array}{c} C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; A \to B) \\ \Delta; \Gamma \vdash e_2 : A \end{array}}{C\, e_2 : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; B)} \qquad \textsc{C-app}_2\quad \frac{\begin{array}{c} \Delta; \Gamma \vdash e_1 : A \to B \\ C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; A) \end{array}}{e_1\, C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; B)}$$

$$\textsc{C-tapp}\quad \frac{C : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; \forall \alpha.\, A) \qquad \mathrm{FV}(B) \subseteq \Delta}{C\langle\rangle : (\Delta' \mid \Gamma'; B') \rightsquigarrow (\Delta \mid \Gamma; A[B/\alpha])}$$

Fig. 9. An excerpt of the grammar of well-typed contexts and their typing rules.

Here, an expression e is said to terminate, written $e \downarrow$, if $(\emptyset, e) \to_{\mathrm{tp}}^{*} (\sigma, v; \vec{e})$ for some final state σ, value v, and additional threads $\vec{e}$—*i.e.*, if a program has e in its main (and initially only) thread, then there there is an execution of that program in which its main thread terminates with a value.

The above definition of contextual refinement extends the standard definition for sequential languages. We follow Turon et al. [2013a] by only taking the termination behavior of the main thread into account, *i.e.*, once the main thread of the implementation has terminated, the main thread of the specification should terminate, too.

At first glance, the definition of contextual refinement might appear much weaker than it actually is since it only talks about termination and not about the resulting values. However, this is not the case. For instance, assuming $\emptyset; \emptyset \vDash e \leq_{ctx} e' : \mathbf{Z}$, we obtain that if computation of e results in some number $n \in \mathbb{Z}$, then so does e'. To see this, simply take the context $\texttt{if}\ [\,] = n\ \texttt{then}\ ()\ \texttt{else}\ \Omega$, where Ω is a program that does not terminate. Similar arguments can be employed to show that contextual refinement implies stronger properties—*e.g.*, that corresponding memory locations in the heap always store indistinguishable values.

8.2 The Binary Value Interpretation

Similar to the unary logical relation for semantic typing, we define the binary logical relation for logical refinements in two stages:

(1) We mutually define the value interpretation $[\![A]\!]_\delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square$ and the expression interpretation $[\![A]\!]_\delta^{\mathrm{e}} : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$, both over *closed* values/expressions, where $\delta : \mathit{Tvar} \rightharpoonup_{\mathrm{fin}} (\mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square)$ is the interpretation for free type variables.

(2) We define the logical refinement relation on *open* terms $\Delta; \Gamma \vDash e \leq_{log} e' : A$ by lifting the value and expression relations to open terms using a closing substitution.

$$
\begin{aligned}
[\![A]\!]^{\mathsf{e}}_{\delta} &\triangleq \lambda\,(e, e').\, \forall j, K.\, \mathrm{SpecCtx} * j \Rrightarrow K[e'] \mathrel{-\!\!*} \mathrm{wp}\ e \left\{v.\, \exists v'.\, j \Rrightarrow K[v'] * [\![A]\!]_{\delta}(v, v')\right\}\\[1ex]
[\![\alpha]\!]_{\delta} &\triangleq \delta(\alpha)\\
[\![\mathbf{1}]\!]_{\delta} &\triangleq \lambda\,(v, v').\, v = v' = ()\\
[\![\mathbf{2}]\!]_{\delta} &\triangleq \lambda\,(v, v').\, v = v' \in \{\mathtt{true}, \mathtt{false}\}\\
[\![\mathbf{Z}]\!]_{\delta} &\triangleq \lambda\,(v, v').\, v = v' \in \mathbb{Z}\\
[\![A_1 \times A_2]\!]_{\delta} &\triangleq \lambda\,(v, v').\, \exists v_1, v_2, v_1', v_2'.\, (v = (v_1, v_2)) * (v' = (v_1', v_2')) * [\![A_1]\!]_{\delta}(v_1, v_1') * [\![A_2]\!]_{\delta}(v_2, v_2')\\
[\![A_1 + A_2]\!]_{\delta} &\triangleq \lambda\,(v, v').\ \textstyle\bigvee_{i \in \{1,2\}} \exists w, w'.\, (v = \mathtt{inj}_i\, w) * (v' = \mathtt{inj}_i\, w') * [\![A_i]\!]_{\delta}(w, w')\\
[\![A \to B]\!]_{\delta} &\triangleq \lambda\,(v, v').\ \square\left(\forall w, w'.\ [\![A]\!]_{\delta}(w, w') \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_{\delta}(v\, w, v'\, w')\right)\\
[\![\forall \alpha.\, A]\!]_{\delta} &\triangleq \lambda\,(v, v').\ \square\left(\forall (\Psi : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\square}).\, [\![A]\!]^{\mathsf{e}}_{\delta, \alpha \mapsto \Psi}(v\langle\rangle, v'\langle\rangle)\right)\\
[\![\exists \alpha.\, A]\!]_{\delta} &\triangleq \lambda\,(v, v').\, \exists (\Psi : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\square}).\\
&\qquad\qquad \exists w, w'.\, (v = \mathtt{pack}\langle w\rangle) * (v' = \mathtt{pack}\langle w'\rangle) * [\![A]\!]_{\delta, \alpha \mapsto \Psi}(w, w')\\
[\![\mu \alpha.\, A]\!]_{\delta} &\triangleq \mu\,(\Psi : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\square}).\\
&\qquad \lambda\,(v, v').\, \exists w, w'.\, (v = \mathtt{fold}\, w) * (v' = \mathtt{fold}\, w') * \triangleright [\![A]\!]_{\delta, \alpha \mapsto \Psi}(w, w')\\
[\![\mathtt{ref}(A)]\!]_{\delta} &\triangleq \lambda\,(v, v').\, \exists \ell, \ell'.\, (v = \ell) * (v' = \ell') * \boxed{\exists w, w'.\, \ell \mapsto w * \ell' \mapsto_{\mathsf{s}} w' * [\![A]\!]_{\delta}(w, w')}^{\mathcal{N}.\ell.\ell'}
\end{aligned}
$$

Fig. 10. The expression interpretation $[\![\_]\!]^{\mathsf{e}}$ and value interpretation $[\![\_]\!]$ for logical refinement in **MyLang**.

The value and expression interpretations are shown in Figure 10. We begin by presenting the former; the latter will be presented in §8.3.

The binary value interpretation is a generalization of its unary counterpart. Values of base types ($\mathbf{1}$, $\mathbf{2}$, and $\mathbf{Z}$) are related if they are equal values of the respective type. Values of the product type are related if both are pairs of values, which are related component-wise by the value interpretations of the corresponding types. Values of the sum type are related if they are both constructed using the same injection with underlying values related at the corresponding type. Values of the function type are related if applying them to values related at the domain type produces expressions related at the codomain type. Values of the universal type are related if their specializations are related, regardless of which (persistent) predicate we take as the value interpretation of the quantified type. Values of the existential type are related if they are both ADTs such that there is a (persistent) predicate for the value interpretation of the quantified type. Values of the recursive type are related if both are a `fold` and their arguments are related one step later. Finally, values of the reference type are related if they are locations that always store related values.

As we did in the unary logical relation, we define the binary logical relation on open expressions using a closing substitution. For that, we first define the interpretation of typing contexts:

$$
\begin{aligned}
[\![\emptyset]\!]^{\mathsf{c}}_{\delta}(\epsilon, \epsilon) &\triangleq \mathrm{True}\\
[\![\Gamma, x : A]\!]^{\mathsf{c}}_{\delta}((\vec{v}, w), (\vec{v'}, w')) &\triangleq [\![\Gamma]\!]^{\mathsf{c}}_{\delta}(\vec{v}, \vec{v'}) * [\![A]\!]_{\delta}(w, w')
\end{aligned}
$$

We then define the binary logical relation for logical refinement $\Delta; \Gamma \vDash e \leq_{log} e' : A$ as follows:

$$
\Delta; \Gamma \vDash e \leq_{log} e' : A \triangleq \square \begin{pmatrix} \forall \delta, \vec{v}, \vec{v'}.\, \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta) \mathrel{-\!\!*} \\ \qquad [\![\Gamma]\!]^{\mathsf{c}}_{\delta}(\vec{v}, \vec{v'}) \mathrel{-\!\!*} [\![A]\!]^{\mathsf{e}}_{\delta}(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}]) \end{pmatrix} \quad \text{where } \vec{x} = \mathrm{dom}(\Gamma)
$$

8.3 The Binary Expression Interpretation

While the binary value interpretation $[\![A]\!]_\delta(v, v')$ is an immediate generalization of the unary version, the binary expression interpretation $[\![A]\!]^{\mathsf{e}}_\delta(e, e')$ requires more work since Iris has no built-in support for relational reasoning.[21] No matter: instead of extending Iris with primitive support for relational reasoning (*e.g.*, a relational version of weakest preconditions), we will show how to use the idea of *specification resources* (due to Turon et al. [2013a]) to *encode* relational reasoning as a derived concept on top of ordinary Iris weakest preconditions.

To explain the idea of specification resources, recall that the expression interpretation $[\![A]\!]^{\mathsf{e}}_\delta(e, e')$ describes a refinement between the *implementation* e and *specification* e'. Intuitively, $[\![A]\!]^{\mathsf{e}}_\delta(e, e')$ says that for each terminating execution of the implementation e, there is a related terminating execution for the specification e' such that $[\![A]\!]_\delta(v, v')$ where v and v' are the values of e and e', respectively. Following the approach by Turon et al. [2013a], this intuitive idea can be expressed using a weakest precondition on the implementation e with a pre- and postcondition that express the existence of a related execution for the specification e'. To describe that the execution of the specification is related to the execution of the implementation, we use *specification resources*:

- The *specification thread connective* $j \Mapsto e$ describes unique ownership of a thread (with thread identifier j) in the specification program, currently executing expression e.
- The *specification points-to connective* $\ell \mapsto_{\mathsf{s}} v$ describes unique ownership of a memory location ℓ in the specification program, currently storing value v.

Like the ghost counter in §7, specification resources are an instance of ghost state—they do not represent ownership of physical resources subject to weakest preconditions, but are rather there strictly for logical purposes. This means that the specification points-to connective $\ell \mapsto_{\mathsf{s}} v$ should not be confused with the ordinary points-to connective $\ell \mapsto v$. The ordinary points-to connective $\ell \mapsto v$ describes ownership of a physical location that appears in the implementation program, whereas the specification points-to connective $\ell \mapsto_{\mathsf{s}} v$ describes ownership of a logical location that appears in the specification program. Like all forms of ghost state in Iris, specification resources can be manipulated using the update modality $\Rrightarrow$. The rules are given in Figure 11. These rules basically express that one can update $j \Mapsto e$ into $j \Mapsto e'$ provided that e steps to e' in the operational semantics. For heap-manipulating operations (allocation, load, store, CAS, and FAA), one additionally has to update ownership of the required specification points-to connectives $\ell \mapsto_{\mathsf{s}} v$. The assertion SpecCtx is there for administrative reasons (which we will discuss below).

Putting all this together, the expression interpretation can be formalized as follows:

$$[\![A]\!]^{\mathsf{e}}_\delta \triangleq \lambda\,(e, e').\, \forall j, K.\, \mathrm{SpecCtx} * j \Mapsto K[e'] \twoheadrightarrow \mathrm{wp}\; e \left\{v.\, \exists v'.\, j \Mapsto K[v'] * [\![A]\!]_\delta(v, v')\right\}$$

This definition reads as follows: assuming a specification thread (with identifier j) contains the expression e' in evaluation position (at context K), then for any execution of the implementation e that results in a value v, there is a related execution from e' to a related value v'. That the related execution obeys the operational semantics is guaranteed by the fact that the only way to manipulate specification resources is through the rules in Figure 11, which exactly correspond to what steps are allowed in the operational semantics.

The definition of specification resources. We now explain how specification resources are defined in Iris. This is done in two steps:

(1) Using Iris's flexible ghost state mechanism, we obtain the connectives $j \Mapsto e$ and $\ell \mapsto_{\mathsf{s}} v$.

[21] Actually, the (unary) weakest-precondition connective in Iris is not built-in either—it is encoded from more primitive constructs. But an exploration of how that works is outside the scope of this paper. See Jung et al. [2018b] for details.

$$\begin{array}{rr}
\mathrm{SpecCtx} * (e_1 \rightarrow_{\mathsf{pure}} e_2) * j \Mapsto K[e_2] \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} j \Mapsto K[e_1] & (\textsc{spec-pure}) \\
\mathrm{SpecCtx} * j \Mapsto K[\texttt{ref}(v)] \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \exists \ell.\, \ell \mapsto_{\mathsf{s}} v * j \Mapsto K[\ell] & (\textsc{spec-alloc}) \\
\mathrm{SpecCtx} * j \Mapsto K[\,!\,\ell] * \ell \mapsto_{\mathsf{s}} v \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \ell \mapsto_{\mathsf{s}} v * j \Mapsto K[v] & (\textsc{spec-load}) \\
\mathrm{SpecCtx} * j \Mapsto K[\ell \leftarrow w] * \ell \mapsto_{\mathsf{s}} v \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \ell \mapsto_{\mathsf{s}} w * j \Mapsto K[()] & (\textsc{spec-store}) \\
\mathrm{SpecCtx} * j \Mapsto K[\texttt{CAS}(\ell, v, w)] * \ell \mapsto_{\mathsf{s}} v \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \ell \mapsto_{\mathsf{s}} w * j \Mapsto K[\texttt{true}] & (\textsc{spec-cas-suc}) \\
\mathrm{SpecCtx} * (v \neq w) * j \Mapsto K[\texttt{CAS}(\ell, w, u)] * \ell \mapsto_{\mathsf{s}} v \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \ell \mapsto_{\mathsf{s}} v * j \Mapsto K[\texttt{false}] & (\textsc{spec-cas-fail}) \\
\mathrm{SpecCtx} * j \Mapsto K[\texttt{FAA}(\ell, m)] * \ell \mapsto_{\mathsf{s}} n \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \ell \mapsto_{\mathsf{s}} (n + m) * j \Mapsto K[n] & (\textsc{spec-faa}) \\
\mathrm{SpecCtx} * j \Mapsto K[\texttt{fork}\ \{e\}\,] \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \exists j'.\, j \Mapsto K[()] * j' \Mapsto e & (\textsc{spec-fork})
\end{array}$$

Fig. 11. Rules for specification resources (we implicitly assume $\mathcal{N}_{\mathsf{spec}} \subseteq \mathcal{E}$).

$$\begin{array}{rr}
\mathrm{SpecCnf}(\sigma, \vec{e}) * j \Mapsto e \vdash e_j = e & (\textsc{sthread-agree}) \\
\mathrm{SpecCnf}(\sigma, \vec{e}) * (j = \mathrm{length}(\vec{e})) \vdash \mathrm{SpecCnf}(\sigma, \vec{e}\, e) * j \Mapsto e & (\textsc{sthread-alloc}) \\
\mathrm{SpecCnf}(\sigma, \vec{e}) * j \Mapsto e \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \mathrm{SpecCnf}(\sigma, \vec{e}[j \mapsto e']) * j \Mapsto e' & (\textsc{sthread-upd}) \\
\mathrm{SpecCnf}(\sigma, \vec{e}) * \ell \mapsto_{\mathsf{s}} v \vdash \sigma(\ell) = v & (\textsc{sheap-agree}) \\
\mathrm{SpecCnf}(\sigma, \vec{e}) * (\ell \notin \mathrm{dom}(\sigma)) \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \mathrm{SpecCnf}(\sigma \uplus \{(\ell, v)\}, \vec{e}) * \ell \mapsto_{\mathsf{s}} v & (\textsc{sheap-alloc}) \\
\mathrm{SpecCnf}(\sigma \uplus \{(\ell, v)\}, \vec{e}) * \ell \mapsto_{\mathsf{s}} v \vdash \mathrel{|\!\!\!\Rrightarrow}_{\mathcal{E}} \mathrm{SpecCnf}(\sigma \uplus \{(\ell, v')\}, \vec{e}) * \ell \mapsto_{\mathsf{s}} v' & (\textsc{sheap-upd}) \\
\mathrm{timeless}(\mathrm{SpecCnf}(\sigma, \vec{e})) \text{ and } \mathrm{timeless}(j \Mapsto e) \text{ and } \mathrm{timeless}(\ell \mapsto_{\mathsf{s}} v) & (\textsc{spec-timeless})
\end{array}$$

Fig. 12. Primitive rules for specification resources.

(2) Using Iris's invariant mechanism, we ensure that these connectives are only manipulated in ways that obey the operational semantics of **MyLang**.

In the first step, we instantiate Iris with a suitable ghost theory (the details of which are beyond the scope of this paper) in order to establish the soundness of a number of primitive proof rules concerning the new connectives $j \Mapsto e$ and $\ell \mapsto_{\mathsf{s}} v$, together with an ephemeral proposition $\mathrm{SpecCnf}(\sigma, \vec{e})$ that keeps track of the entire heap σ and the entire thread-pool $\vec{e}$ of the specification. These primitive rules are shown in Figure 12. Using these rules, one can basically manipulate $j \Mapsto e$ and $\ell \mapsto_{\mathsf{s}} v$ as long as that is done in sync with $\mathrm{SpecCnf}(\sigma, \vec{e})$. The fact that $\mathrm{SpecCnf}(\sigma, \vec{e})$ is in sync is witnessed by the rules sthread-agree and sheap-agree, which say that if we own the thread connective $j \Mapsto e$ (respectively, the points-to connective $\ell \mapsto_{\mathsf{s}} v$), then the thread j is in fact in the thread-pool $\vec{e}$ (respectively, the location ℓ is in the heap σ), where it is mapped to e (respectively, v). When allocating or updating a thread connective $j \Mapsto e$ (using sthread-alloc and sthread-upd) or a points-to connective $\ell \mapsto_{\mathsf{s}} v$ (using sheap-alloc and sheap-upd), one has to change $\mathrm{SpecCnf}(\sigma, \vec{e})$ in a corresponding fashion.

In the second step, we use Iris's invariant mechanism to ensure that the thread and points-to connectives are manipulated in a way that obeys the operational semantics. For that, we use the

RS-VAR
$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vDash x \leq_{log} x : A}$$

RS-UNIT
$$\Delta; \Gamma \vDash () \leq_{log} () : \mathbf{1}$$

RS-BOOL
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Delta; \Gamma \vDash b \leq_{log} b : \mathbf{2}}$$

RS-INT
$$\frac{n \in \mathbb{Z}}{\Delta; \Gamma \vDash n \leq_{log} n : \mathbf{Z}}$$

RS-REC
$$\frac{\Delta; \Gamma, x : A, f : A \to B \vDash e \leq_{log} e' : A}{\Delta; \Gamma \vDash (\texttt{rec}\, f(x) = e) \leq_{log} (\texttt{rec}\, f(x) = e') : A \to B}$$

RS-APP
$$\frac{\Delta; \Gamma \vDash e_1 \leq_{log} e_1' : A \to B \qquad \Delta; \Gamma \vDash e_2 \leq_{log} e_2' : A}{\Delta; \Gamma \vDash e_1\, e_2 \leq_{log} e_1'\, e_2' : B}$$

Fig. 13. An excerpt of relational semantic typing rules (compatibility lemmas).

following invariant:

$$\mathrm{SpecInv}(\sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}}) \triangleq \boxed{\exists \sigma, \vec{e}.\, \mathrm{SpecCnf}(\sigma, \vec{e}) * \Big((\sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}}) \to_{\mathsf{tp}}^{*} (\sigma, \vec{e})\Big)}^{\mathcal{N}_{\mathsf{spec}}}$$
$$\mathrm{SpecCtx} \triangleq \exists \sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}}.\, \mathrm{SpecInv}(\sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}})$$

Given some initial heap σ_{init} and initial thread-pool $\vec{e}_{\mathsf{init}}$, the invariant $\mathrm{SpecInv}(\sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}})$ expresses that the heap and thread-pool in $\mathrm{SpecCnf}(\sigma, \vec{e})$ can always be reached by taking a sequence of steps in the operational semantics from $(\sigma_{\mathsf{init}}, \vec{e}_{\mathsf{init}})$. Most of the time, with the exception of the soundness proof in §8.5, we do not need to know the initial state. Hence, we define SpecCtx, which existentially quantifies the initial state.

With the above definitions in hand, we can now prove all the rules in Figure 11. These follow from Iris's rules for invariants, together with the primitive rules for specification resources in Figure 12. Since specification resources are timeless, we can use the rule INV-OPEN-UPD-TL to open the invariant SpecInv without a later modality. Note that, due to the use of an invariant to define specification resources in Iris, we need the premise SpecCtx and side-condition $\mathcal{N}_{\mathsf{spec}} \subseteq \mathcal{E}$ in the rules in Figure 11.

8.4 Compatibility Lemmas

Just as we proved semantic typing rules for the unary logical relation in §6, we now prove *relational* semantic typing rules for **MyLang**. In logical relations jargon, the relational semantic typing rules are often referred to as *compatibility lemmas*, see, *e.g.*, [Pitts 2005], since they show how the binary logical relation is "compatible" with the various constructs of **MyLang**. A selection of the compatibility lemmas for **MyLang** are presented in Figure 13. Below we discuss the proofs of a few of them. The proofs of other compatibility lemmas follow in a similar fashion, just as many of the semantic typing rules in §6 followed a common essential structure.

Before we go on to discuss some of the compatibility lemmas, we prove the monadic rules for the binary expression relation, which are a generalization of the unary versions in Lemma 6.1

LEMMA 8.2 (THE MONADIC RULES FOR THE EXPRESSION INTERPRETATION).

$$[\![A]\!]_\delta(v, v') \mathrel{-\!\!*} [\![A]\!]^{\mathsf{e}}_\delta(v, v') \qquad \text{(BIN-VAL)}$$
$$[\![A]\!]^{\mathsf{e}}_\delta(e, e') * \big(\forall v, v'.\, [\![A]\!]_\delta(v, v') \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(K[v], K'[v'])\big) \mathrel{-\!\!*} [\![B]\!]^{\mathsf{e}}_\delta(K[e], K'[e']) \qquad \text{(BIN-BIND)}$$

Proof. The rule bin-val follows immediately from wp-val. The following is a proof tree for bin-bind (in which we employ the shorthand $F \triangleq \forall v, v'.\ \llbracket A \rrbracket_\delta(v, v') \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_\delta(K[v], K'[v'])$ and $\Phi \triangleq \lambda w.\ \exists w'.\ j \Mapsto K''[w'] * \llbracket B \rrbracket_\delta(w, w')$), and K'' is arbitrary, *i.e.*, universally quantified):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{SpecCtx} * j \Mapsto K''[K'[v']] * \llbracket B \rrbracket^{\mathrm{e}}_\delta(K[v], K'[v']) \vdash \mathsf{wp}\ K[v]\ \{\Phi\}}\ \text{unfold } \llbracket B \rrbracket^{\mathrm{e}} \text{ on LHS, } \forall\text{-elim, } {-\!\!*}\text{-ELIM}}{\mathrm{SpecCtx} * j \Mapsto K''[K'[v']] * \llbracket A \rrbracket_\delta(v, v') * F \vdash \mathsf{wp}\ K[v]\ \{\Phi\}}\ \text{unfold } F \text{ on LHS, } \forall\text{-elim, } {-\!\!*}\text{-ELIM}}{\mathrm{SpecCtx} * \mathsf{wp}\ e\ \{v.\ \exists v'.\ j \Mapsto K''[K'[v']] * \llbracket A \rrbracket_\delta(v, v')\} * F \vdash \mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ K[v]\ \{\Phi\}\}}\ \text{WP-WAND, } \exists\text{-elim}}{\mathrm{SpecCtx} * \mathsf{wp}\ e\ \{v.\ \exists v'.\ j \Mapsto K''[K'[v']] * \llbracket A \rrbracket_\delta(v, v')\} * F \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}\ \text{WP-BIND}}{\mathrm{SpecCtx} * j \Mapsto K''[K'[e]] * \llbracket A \rrbracket^{\mathrm{e}}_\delta(e, e') * F \vdash \mathsf{wp}\ K[e]\ \{\Phi\}}\ \text{unfold } \llbracket A \rrbracket^{\mathrm{e}} \text{ on LHS, } \forall\text{-elim, } {-\!\!*}\text{-ELIM}}{\llbracket A \rrbracket^{\mathrm{e}}_\delta(e, e') * F \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta(K[e], K'[e'])}\ \begin{array}{l}\text{unfold } \llbracket B \rrbracket^{\mathrm{e}} \text{ on RHS,}\\ \forall\text{-intro, } {-\!\!*}\text{-INTRO}\end{array}$$

Note that the invariant SpecCtx is persistent and hence duplicable. In the proof tree above, we have elided the steps corresponding to duplicating this invariant, as well as applications of basic structural reasoning on evaluation contexts, such as $K''[K'[e]] = (K'' \circ K')[e]$. □

The proof of RS-var. By unfolding the definition of the logical refinement relation, we have to prove $\llbracket \Gamma \rrbracket^{\mathrm{c}}_\delta(\vec{v}, \vec{v'}) \mathrel{-\!\!*} \llbracket A \rrbracket^{\mathrm{e}}_\delta(x[\vec{v}/\vec{x}], x[\vec{v'}/\vec{x}])$. This result follows from the premise $x : A \in \Gamma$ and the rule bin-val.

The proof of RS-app. In order to prove the compatibility lemma RS-app, we prove the following auxiliary result for closed expressions, from which the semantic typing rule on open expressions easily follows:

$$\llbracket A \to B \rrbracket^{\mathrm{e}}_\delta(e_1, e_1') * \llbracket A \rrbracket^{\mathrm{e}}_\delta(e_2, e_2') \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_\delta((e_1\ e_2), (e_1'\ e_2'))$$

Below there follows a proof tree for the auxiliary result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\llbracket A \rrbracket_\delta(v_2, v_2') \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ v_2), (v_1'\ v_2'))) * \llbracket A \rrbracket_\delta(v_2, v_2') \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ v_2), (v_1'\ v_2'))}\ {-\!\!*}\text{-ELIM on LHS}}{\square\,(\forall w, w'.\ \llbracket A \rrbracket_\delta(w, w') \mathrel{-\!\!*} \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ w), (v_1'\ w'))) * \llbracket A \rrbracket_\delta(v_2, v_2') \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ v_2), (v_1'\ v_2'))}\ \square\text{-ELIM, } \forall\text{-elim on LHS}}{\llbracket A \to B \rrbracket_\delta(v_1, v_1') * \llbracket A \rrbracket_\delta(v_2, v_2') \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ v_2), (v_1'\ v_2'))}\ \text{unfold } \llbracket A \to B \rrbracket^{\mathrm{e}} \text{ on LHS}}{\llbracket A \to B \rrbracket_\delta(v_1, v_1') * \llbracket A \rrbracket^{\mathrm{e}}_\delta(e_2, e_2') \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta((v_1\ e_2), (v_1'\ e_2'))}\ \text{BIN-BIND}}{\llbracket A \to B \rrbracket^{\mathrm{e}}_\delta(e_1, e_1') * \llbracket A \rrbracket^{\mathrm{e}}_\delta(e_2, e_2') \vdash \llbracket B \rrbracket^{\mathrm{e}}_\delta((e_1\ e_2), (e_1'\ e_2'))}\ \text{BIN-BIND}$$

This proof tree is similar to the one for the semantic typing rule S-app in the unary case. Reading this proof tree bottom-up, we start by using bin-bind twice (following the scheme we described in §6.3), first for expressions e_1 and e_1' in contexts $K \triangleq [\,]\ e_2$ and $K' \triangleq [\,]\ e_2'$, and then for expressions e_2 and e_2' in contexts $K \triangleq v_1\ [\,]$ and $K' \triangleq v_1'\ [\,]$. The last step again demonstrates why "logical relations" are called "logical"—we use Iris's modus ponens rule $(Q \mathrel{-\!\!*} R) * Q \vdash R$ (${-\!\!*}$-elim) to eliminate the magic wand that appears in the interpretation of the function type $A \to B$.

The actual compatibility lemma RS-app follows from this auxiliary lemma in the same way that S-app followed from its corresponding auxiliary result in §6.4.

8.5 The Fundamental Theorem and Soundness

Theorem 8.3 (Fundamental theorem of binary logical relations). *Well-typed terms are related to themselves, i.e., if* $\Delta; \Gamma \vdash e : A$ *then* $\Delta; \Gamma \vDash e \leq_{log} e : A$.

Proof. By straightforward induction on the typing derivation $\Delta; \Gamma \vdash e : A$. For each case in the induction proof, we use the corresponding compatibility lemma. □

Lemma 8.4 (Congruency of binary logical relations). *The binary logical relation is closed under well-typed program contexts, i.e., if* $\Delta; \Gamma \vDash e \leq_{log} e' : A$ *and* $C : (\Delta \mid \Gamma; A) \rightsquigarrow (\Delta' \mid \Gamma'; A')$ *then* $\Delta'; \Gamma' \vDash C[e] \leq_{log} C[e'] : A'$.

Proof. By straightforward induction on the derivation of $C : (\Delta \mid \Gamma; A) \rightsquigarrow (\Delta' \mid \Gamma'; A')$. In each case, we apply the appropriate compatibility lemma and, when necessary, use the fundamental theorem (Theorem 8.3) to show that well-typed expressions are related to themselves. □

Lemma 8.5 (Adequacy of the binary logical relations). *The binary logical relation preserves termination, i.e., if $\emptyset; \emptyset \vDash e \leq_{log} e' : A$, then $e \downarrow$ implies $e' \downarrow$.*

Proof. In order to prove this lemma we make use of adequacy of Iris's weakest preconditions [Krebbers et al. 2017a]. For brevity's sake, we do not consider Iris's adequacy statement in its full generality, but rather consider a version that is instantiated with the ghost theory for specification resources. That is, given a first-order proposition ϕ and a proof of

$$\mathrm{SpecCnf}(\emptyset, \emptyset) \vdash \mathrm{wp}\ e\ \{\phi\},$$

if $e \downarrow$, then ϕ holds at the meta-level.

To prove our lemma, we pick $\phi \triangleq e' \downarrow$, which means we are done once we have proved $\mathrm{SpecCnf}(\emptyset, \emptyset) \vdash \mathrm{wp}\ e\ \{e' \downarrow\}$. We prove this result in the following steps:

$$\begin{aligned} \mathrm{SpecCnf}(\emptyset, \emptyset) &\vdash {\mid\!\Rrightarrow}\ \mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto e' && (\textsc{step1}) \\ \mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto e' &\vdash \mathrm{wp}\ e\ \big\{v.\ \exists v'.\ 1 \Mapsto v' * [\![A]\!]_\delta(v, v')\big\} && (\textsc{step2}) \\ \mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto v' &\vdash {\mid\!\Rrightarrow}\ e' \downarrow && (\textsc{step3}) \end{aligned}$$

In step1, we allocate the invariant $\mathrm{SpecInv}(\emptyset, e')$. We do this by first creating a specification thread resource $1 \Mapsto e'$ for the main thread (using sthread-alloc), and then transferring ownership of $\mathrm{SpecCnf}(\emptyset, e')$ into the invariant $\mathrm{SpecInv}(\emptyset, e')$ (using inv-alloc).

In step2, we make use of our premise $\emptyset; \emptyset \vDash e \leq_{log} e' : A$. Since we are considering closed programs, by definition of the binary logical relation this premise is equivalent to $[\![A]\!]^{\mathsf{e}}_\delta(e, e')$. By unfolding the expression interpretation we then get:

$$\forall j, K.\ \mathrm{SpecInv}(\emptyset, e') * j \Mapsto K[e'] \mathrel{-\!\!*} \mathrm{wp}\ e\ \big\{v.\ \exists v'.\ j \Mapsto K[v'] * [\![A]\!]_\delta(v, v')\big\}$$

Our result is obtained by specializing this statement by picking $K = [\,]$ and $j = 1$:

$$\mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto e' \mathrel{-\!\!*} \mathrm{wp}\ e\ \big\{v.\ \exists v'.\ 1 \Mapsto v' * [\![A]\!]_\delta(v, v')\big\}$$

In step3, we open the invariant $\mathrm{SpecInv}(\emptyset, e')$ (using inv-open-upd-tl) to obtain that we have $(\emptyset, e') \to^*_{\mathrm{tp}} (\sigma, \vec{e})$ for some heap σ and threadpool $\vec{e}$ with $\mathrm{SpecCnf}(\sigma, \vec{e})$. Since we have $1 \Mapsto v'$, we obtain that the main thread of $\vec{e}$ is the value v' (by sthread-agree), which gives $e' \downarrow$ as desired.

The proof tree below shows how these steps lead to the final result:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto v' * [\![A]\!]_\delta(v, v') \vdash {\mid\!\Rrightarrow}\ e' \downarrow}\ \textsc{step3}}{\mathrm{SpecInv}(\emptyset, e') * \mathrm{wp}\ e\ \big\{v.\ \exists v'.\ 1 \Mapsto v' * [\![A]\!]_\delta(v, v')\big\} \vdash \mathrm{wp}\ e\ \{e' \downarrow\}}\ \textsc{wp-wand},\ \exists\text{-elim},\ \textsc{wp-}{\mid\!\Rrightarrow}}{\mathrm{SpecInv}(\emptyset, e') * 1 \Mapsto e' \vdash \mathrm{wp}\ e\ \{e' \downarrow\}}\ \textsc{step2}}{\mathrm{SpecCnf}(\emptyset, \emptyset) \vdash \mathrm{wp}\ e\ \{e' \downarrow\}}\ \textsc{step1},\ {\mid\!\Rrightarrow}\textsc{-wp},\ {\mid\!\Rrightarrow}\textsc{-mono}$$

Note that, once established, we can keep the invariant $\mathrm{SpecInv}(\emptyset, e')$ around throughout the proof since it is persistent (and thus duplicable). □

Theorem 8.6 (Soundness of binary logical relations). *The binary logical relation is sound w.r.t. contextual refinement, i.e., if $\Delta; \Gamma \vdash e : A$ and $\Delta; \Gamma \vdash e' : A$, then $\Delta; \Gamma \vDash e \leq_{log} e' : A$ implies $\Delta; \Gamma \vDash e \leq_{ctx} e' : A$.*

```
stack$_{fg}$ ≜ Λ.
  let s = ref(ref(None)) in
  let push$_{fg}$ = λ x.
    let z = ! s in
    if CAS(s, z, ref(Some(x, fold z))) then ()
    else push$_{fg}$ x in
  let pop$_{fg}$ = λ ().
    let z = ! s in
    match ! z with
      None          ⇒ None
    | Some(hd, tl) ⇒
        if CAS(s, z, unfold tl) then Some(hd)
        else pop$_{fg}$ ()
    end in
  (push$_{fg}$, pop$_{fg}$)
```

```
stack$_{cg}$ ≜ Λ.
  let s = ref(None) in
  let l = newlock () in
  let push$_{cg}$ = λ x.
    acquire l;
    s ← Some(x, fold ! s);
    release l in
  let pop$_{cg}$ = λ ().
    acquire l;
    let mx =
      match ! z with
        None          ⇒ None
      | Some(hd, tl) ⇒
          s ← unfold tl; Some(hd)
      end in
    release l; mx in
  (push$_{cg}$, pop$_{cg}$)
```

Fig. 14. The source code of a fine-grained (left) and coarse-grained (right) concurrent stack.

$$j \Rightarrow K[\mathsf{newlock}\ ()] \vdash \Rrightarrow_{\mathcal{E}} \exists l.\ \mathsf{isLock_s}(l, \mathsf{false}) * j \Rightarrow K[l] \qquad (\textsc{spec-newlock})$$

$$\mathsf{isLock_s}(l, \mathsf{false}) * j \Rightarrow K[\mathsf{acquire}\ l] \vdash \Rrightarrow_{\mathcal{E}} \mathsf{isLock_s}(l, \mathsf{true}) * j \Rightarrow K[()] \qquad (\textsc{spec-acquire})$$

$$\mathsf{isLock_s}(l, \mathsf{true}) * j \Rightarrow K[\mathsf{release}\ l] \vdash \Rrightarrow_{\mathcal{E}} \mathsf{isLock_s}(l, \mathsf{false}) * j \Rightarrow K[()] \qquad (\textsc{spec-release})$$

$$\mathsf{timeless}(\mathsf{isLock_s}(l, b)) \qquad (\textsc{spec-lock-timeless})$$

Fig. 15. Rules for lock specification resources.

Proof. By definition of contextual refinement, in order to prove $\Delta; \Gamma \vDash e \preceq_{ctx} e' : A$, we are given a well-typed program context $C : (\Delta \mid \Gamma; A) \rightsquigarrow (\emptyset \mid \emptyset; \mathbf{1})$ and have to show that $C[e] \downarrow$ implies $C[e'] \downarrow$. By the assumption and congruency (Lemma 8.4), we have $\emptyset; \emptyset \vDash C[e] \preceq_{log} C[e'] : \mathbf{1}$, which gives that $C[e] \downarrow$ implies $C[e'] \downarrow$ by adequacy (Lemma 8.5). □

8.6 Representation Independence Proofs

The soundness theorem of our binary logical relation (Theorem 8.6) allows us to prove contextual refinement by means of logical refinement. As our logical relation is formalized on top of Iris, we have the entire power and support of Iris at our disposal when proving contextual refinement by means of logical refinement. In this subsection, we demonstrate this power by proving representation independence of two implementations of a concurrent stack displayed in Figure 14. Specifically, we prove the following refinement:

$$\vDash \mathsf{stack_{fg}} \preceq_{ctx} \mathsf{stack_{cg}} : \forall \alpha.\ (\alpha \to \mathbf{1}) \times (\mathbf{1} \to \mathsf{option}(\alpha))$$

The stack ADT provides functions push and pop for pushing and popping elements on and off a stack. Since the stack is dynamically sized, the function push will always succeed. The function pop may fail by returning None if the stack is empty. Here, the option type is defined in the usual way using sums, *i.e.,* $\texttt{option}(A) \triangleq \mathbf{1} + A$, and the constructors are defined as $\texttt{None} \triangleq \texttt{inj}_1\, ()$ and $\texttt{Some}\ v \triangleq \texttt{inj}_2\, v$.

The two implementations of the stack ADT differ in the *granularity* of their concurrency: the first is *fine-grained*—it enforces atomicity at the level of individual instructions—whereas the second is *coarse-grained*—it enforces atomicity via a critical section, protected by a lock.

Concretely, the fine-grained implementation $\texttt{stack}_{\text{fg}}$ employs a private reference s that points to the head of a linked list defined using the following recursive type:

$$\texttt{linkedlist}(A) \triangleq \mu\alpha.\, \texttt{ref}(\texttt{option}(A \times \alpha))$$

The fine-grained implementation uses a technique known as optimistic concurrency to implement push and pop. It first reads the head reference s to the linked list. It then tries to update the head reference using the (atomic) compare-and-set instruction (CAS) to make sure it has not been modified in the meantime. If the CAS fails, another thread must have augmented the reference to the list; in that case, the operation simply starts over, trying to perform the push or pop again.

The coarse-grained implementation $\texttt{stack}_{\text{cg}}$ simply stores the entire stack as a private reference s to a functional list defined using the following recursive type:

$$\texttt{list}(A) \triangleq \mu\alpha.\, \texttt{option}(A \times \alpha)$$

The coarse-grained implementation uses a lock l to make sure the push and pop instructions are carried out atomically. The operation newlock creates a new lock, which is initially in the unlocked state. The lock can be moved into the locked state using the acquire operation, which will block if another thread holds the lock. The lock can be put back into the unlocked state using the release operation. Release does not block, because only if one acquired the lock, it should release the lock.

Although the language **MyLang** does not have locks as primitives, they can easily be implemented using for example a spin lock or a ticket lock. For the purpose of this paper, it does not matter what lock implementation is used—all that matters is that the implementation enjoys the logical rules in Figure 15. (See [Frumin et al. 2021b, §5] for a proof that a spin lock implementation satisfies these logical rules.) Note that since locks are used for the specification side of the refinement, we have only included the rules in terms of specification resources, and not those in terms of weakest preconditions. Furthermore, note that the lock rules are similar to the rules for the heap operations we have seen in Figure 11, but they involve a new predicate $\text{isLock}_{\text{s}}(l, b)$, where $b = \texttt{false}$ means that the lock l is in the unlocked state, and $b = \texttt{true}$ means it is in the locked state.

The proof of the stack refinement. In order to prove contextual refinement of the lock implementations, it suffices, by the soundness of the binary logical relations (Theorem 8.6), to prove the following, corresponding logical refinement:

$$\vDash \texttt{stack}_{\text{fg}} \preceq_{log} \texttt{stack}_{\text{cg}} : \forall\alpha.\, (\alpha \to \mathbf{1}) \times (\mathbf{1} \to \texttt{option}(\alpha))$$

The proof follows the same structure as the proof of safe encapsulation of the symbol ADT in §7. We unfold the definition of the logical refinement judgment, and prove Iris weakest preconditions for the functions push and pop. The crux of the proof involves defining an invariant that relates the internal data structures used in both implementations. Since the stack ADT is polymorphic, this invariant should make sure that the values of both stacks are related by the binary value interpretation corresponding to the type α, which we call $\Phi : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$. To relate the internal data structures of both implementations we define the following Iris propositions:

$$
\begin{aligned}
\bar{\Phi} \triangleq \mu \bar{\Phi} : (\mathit{Val} \times \mathit{Val} \longrightarrow \mathit{iProp}).\, \lambda\, (\ell, v').& \\
&(\ell \mapsto \mathsf{None} * v' = \mathsf{None}) \vee \\
&(\exists w, w', \ell_{tl}, v'_{tl}.\, \ell \mapsto \mathsf{Some}(w, \texttt{fold}\, \ell_{tl}) * v' = \mathsf{Some}(w', \texttt{fold}\, v'_{tl}) * \Phi(w, w') * \triangleright \bar{\Phi}(\ell'_{tl}, v'_{tl})) \\
I \triangleq \boxed{\exists \ell, v'.\, s \mapsto \ell \; * \; s' \mapsto_{\mathsf{s}} v' \; * \; \bar{\Phi}(\ell, v') \; * \; \mathsf{isLock}_{\mathsf{s}}(l', \texttt{false})}^{\mathcal{N}_{\mathsf{stk}}}&
\end{aligned}
$$

In prose, the invariant I states that:

- the private reference s of $\texttt{stack}_{\mathrm{fg}}$ always points to the head of a linked list ℓ;
- the private reference s' of $\texttt{stack}_{\mathrm{cg}}$ always points to a functional list v';
- the values stored in the linked list at ℓ and function list v' are related by Φ; and
- the lock of $\texttt{stack}_{\mathrm{cg}}$ is in unlocked state.

The relation $\bar{\Phi}(\ell, v')$ ensures that the linked list pointed by ℓ and the functional list v' have the same length and that the values in these lists are related by the value interpretation Φ. Note that, as far as the behavior of the ADTs is considered, the lock is never *observed* in the locked state. This is because all the `acquire` statements in the operations of the coarse-grained stack are followed by a `release`, and these operations are all executed atomically.

With the invariant in hand, the proof of the logical refinement is straightforward but lengthy. After we have allocated the resources of the stacks (the lists and locks), we create the Iris invariant I. Subsequently, we prove the refinements of push and pop using Löb induction, where in each step we open and close the invariant I. A detailed and formal proof can be found in the accompanying Coq formalization (see §10 for the URL to the online repository with the Coq formalization).

Summary. We have shown how our logical relation can be used to show representation independence of two implementations of a concurrent stack. We note that since the coarse-grained stack uses locks to sequentialize accesses to the stack, one can understand our logical relations proof of contextual refinement as an alternative to the *linearizability* proof method for concurrent objects [Herlihy and Wing 1990]. A possible advantage of the logical relations approach shown here is that it also applies, *mutatis mutandis*, when the data structure in question involves higher-order functions; for example, one can easily extend the logical relations proof above to the case where the fine-grained and coarse-grained concurrent stacks include a higher-order iterator method. In contrast, linearizability has so far mostly been developed for first-order languages, although recent work by Murawski and Tzevelekos [Murawski and Tzevelekos 2019] has extended linearizability to higher-order programming languages.

9 ADDITIONAL RELATED WORK

The "logical approach to type soundness" that we have advanced in this paper descends directly from multiple lines of prior work on the semantics of higher-order, imperative, and concurrent programming languages. In §4.2, we discussed earlier work on semantic type soundness and step-indexed models. In this section, we briefly survey some other key influences on our work, as well as closely related approaches.

Relational logics for richly-typed languages. The most direct ancestor of our approach is the line of work on *relational logics*—logics for reasoning more abstractly about relational program properties such as parametricity and representation independence in richly-typed languages. The primogenitor of this line is the seminal paper of Plotkin and Abadi [1993], who showed how to define logical relations for a polymorphic programming language in a second-order relational logic. Their approach was extended by Dreyer et al. [2009, 2011], who integrated the "later" (▷)

modality into a Plotkin-Abadi style logic in order to define step-indexed logical relations for a language with polymorphic and recursive types. Dreyer et al.'s motivation was precisely to avoid the tedious step-indexed arithmetic that they had previously experienced when working directly with step-indexed models. Their method was extended further by Dreyer et al. [2010] to handle general (higher-typed) mutable references, using a second-order *relational separation logic*, inspired by [Yang 2007], with a notion of invariants (called "islands", based on prior work of Ahmed et al. [2009]). Turon et al. [2013a] later extended the logical approach to a language with concurrency (using a pre-Iris second-order *concurrent* separation logic called CaReSL), and Krogh-Jespersen et al. [2017] extended it (using Iris) to account for a region-based type-and-effect system.

Though directly continuing this line of work, our "logical approach to type soundness" goes beyond the aforementioned prior pre-Iris work on relational logics in several ways. First of all, we have shown that the approach of building logical relations in separation logic is useful not only in proving relational properties like representation independence but also in formalizing *semantic type soundness* results (a unary property) for richly-typed languages; and we have also demonstrated the utility of the resulting semantic soundness theorems for verifying safe encapsulation of unsafe features. Second, our approach leverages a more modern concurrent separation logic, namely Iris, which offers a richer, more evolved, and still actively evolving logical language in which to encode logical relations models of types. (Iris is also a language-agnostic framework, which can be instantiated for a wide variety of different languages so long as they can be formalized with a relatively standard style of operational semantics [Jung et al. 2018b, §7.3].) Last but not least, thanks to the Iris Proof Mode [Krebbers et al. 2017b], our approach has the key benefit of making it feasible to (fairly rapidly) develop both semantic soundness and representation independence proofs that are fully machine-checked in Coq. In contrast, none of the aforementioned prior work was formally mechanized in a proof assistant.

These benefits of the "logical approach" have already been demonstrated in a significant and growing set of papers that employ it for both unary and relational reasoning (see §10 for citations)—but as we noted in the Introduction, these papers are not always the easiest on-ramps for newcomers wanting to learn the essential methodology. We hope that the present paper helps to fill the pedagogical gap by presenting the logical approach from first principles and in the setting of a simpler (yet still expressive) programming language.

"Semantic soundness" for compilers. A second key ancestor of our work is that of Benton and collaborators [Benton 2006; Benton and Zarfaty 2007; Benton and Tabareau 2009; Benton and Hur 2009]. Over a series of papers, they propounded the idea that "compiler correctness" ought to account for preservation of (source-level) relational reasoning down to the assembly level—and that, to realize this idea, one should build semantic models of high-level types as (binary) relational specifications on low-level code. Though superficially distinct from the kinds of results we have established in this paper, Benton et al.'s compiler correctness theorems were referred to as "semantic soundness" theorems, and indeed there is a strong kinship between theirs and ours. In particular, like our logical-relations models, theirs (1) were formulated using logical abstractions such as the later modality and separating conjunction to support higher-level reasoning, (2) were specifically used to verify that low-level, potentially unsafe code is well-behaved according to the semantic contracts of high-level types, and (3) were formalized in Coq.

Aside from the specific intended application, a key difference between our work and Benton et al.'s is that, although Benton et al.'s models make use of higher-level logical abstractions, the proofs about them are still conducted directly in the model of propositions (rather than in a *bona fide* logic like Iris) and without the rich tactical support for separation logic that the Iris Proof Mode provides, thus rendering them considerably lower-level than ours. This is quite understandable, given that Benton et al.'s work was conducted *before* a number of major advances in (higher-order concurrent) separation logic, which ultimately culminated in the development of Iris. In other words, the work was ahead of its time. Nevertheless, it was a source of inspiration for us in how it used logical relations, along with techniques from step-indexing and separation logic, to carve out "well-behavedness" conditions on potentially unsafe code. Also inspiring to us were Benton et al.'s observations concerning the limitations of syntactic type soundness, which were rather iconoclastic given the predominance of the syntactic approach at the time.

Simulation-based approaches. Around the same time as step-indexed models were being developed in the mid-2000s, there emerged an impressive series of papers on *(bi-)simulation* techniques for relational reasoning in higher-order, imperative, and concurrent languages—*e.g.,* [Koutavas and Wand 2006; Sumii and Pierce 2007; Støvring and Lassen 2007; Lassen and Levy 2007; Sumii 2009]. We still lack a precise understanding of the relationship between logical relations and simulation-based methods—there are tradeoffs in terms of convenience of proof effort—but suffice it to say that, in terms of expressive power, both classes of techniques have proven capable (in principle) of supporting sophisticated relational reasoning in a range of different programming languages. (For more details, we refer the reader to Hur et al. [2012].)

There are, however, a number of differences between the simulation-based methods and the methods we have presented in this paper. First, since the simulation-based methods rely on coinduction (rather than step-indexing) to achieve reasoning about "circular" features (*e.g.,* recursive types, higher-order state), they do not require anything comparable to the tedious reasoning about step-index arithmetic that we remarked upon in §4.3. However, as with direct reasoning in step-indexed models, the simulation-based methods *do* involve explicit, and sometimes low-level, reasoning about the global machine state and invariants on it (see points 2 and 3 in §4.3). Second, nearly all of the work on simulations has been focused exclusively on proving relational properties, not semantic type soundness. One exception is Sumii [2010], who explores the applicability of simulation-based methods to proving safe encapsulation of potentially unsafe deallocation operations in a sequential, *untyped* λ-calculus with higher-order state, but his approach has not seemingly been applied to a wider variety of languages. Lastly, with the exception of the line of work on "parametric simulations" [Hur et al. 2012; Neis et al. 2015], none of the simulation-based approaches have, to our knowledge, been formally mechanized in a proof assistant.

Syntactic type abstraction. In §3, we discussed the strengths and limitations of the syntactic approach to type soundness, noting in particular that syntactic type soundness has nothing to say about whether a language properly supports data abstraction. There is, however, at least one paper proposing a syntactic approach to reasoning about data abstraction properties of ADTs, namely that of Grossman et al. [2000]. Their approach involves introducing a syntactic notion of "principals" into their operational semantics in order to track which values arise from the implementation of an ADT vs. from its client. Although they develop their method in the presence of a wide range of features (including higher-order state), they only use it to prove a limited class of results: one stating that a value of some abstract type must have arisen from calling a specific operation provided by an ADT, and another stating that changing the integer representing a value of some abstract type will not affect client code. The proofs of those results eschew the complexities of semantic models but supplant them with arguments concerning highly intricate syntactic invariants (see for instance the proof of Theorem 3.13 in their paper). Moreover, it is not at all clear how one could apply their method to verify either the `symbol` ADT example from §7 or the representation independence example from §8.6, since those examples involve more complex invariants on state.

Hybrid syntactic/semantic approaches to type soundness. There have been a few approaches to type soundness that incorporate a hybrid of syntactic and semantic/logical elements.

Tofte [1990] proposed an early approach to type inference (and type soundness) for an ML-like language combining polymorphism and mutable references. Tofte's approach, which pre-dates the "progress and preservation" approach [Wright and Felleisen 1994; Harper 2016], defines a semantic typing relation, albeit using coinduction to handle circularities in the construction (rather than step-indexing as we do), and using a syntactic heap typing to track the types of memory locations (rather than a semantic/logical model of heap typing as we formalize with Iris invariants). Tofte's approach was adopted by several others in the early 1990s [Leroy and Weis 1991; Talpin and Jouvelot 1994], when a number of researchers were investigating how best to make ML-style type inference play well with mutable references. However, it fell out of favor after much simpler methods were proposed: the "value restriction" [Wright 1995] for safely combining ML-style polymorphism with references (which was ultimately integrated into both Standard ML and OCaml), and progress and preservation for proving type soundness. Moreover, Tofte's approach was limited to predicative polymorphism (see the discussion in [Tofte 1990, p. 21]), which neither syntactic nor logical type soundness are; and due to its reliance on syntactic techniques, it suffers from the same limitations of syntactic type soundness that we laid out in §3.

Mezzo [Balabonski et al. 2016] is a recently proposed programming language with (broadly speaking) similar goals to Rust: supporting low-level, fine-grained control over the representation and access of data in memory, while preserving type and memory safety. Also like Rust, Mezzo employs a substructural type system in order to track aliasing and ownership of memory. The soundness proof for Mezzo is clearly syntactic, following the tradiiton of progress-and-preservation proofs. However, in order to support a more modular presentation of the soundness proof, Balabonski et al. formalize a notion of "resource" using something called a "monotonic resource algebra" (which is closely related to, but not the same as, the "cameras" and "resource algebras" used in Iris—see the discussion in [Jung et al. 2018b, §9.3]). These resources definitely give their soundness proof a separation logic flavor; yet it remains syntactic, and thus does not offer a way to reason about data abstraction or safe encapsulation of unsafe features.

10 CONCLUSION

In this paper, we have demonstrated that semantic type soundness is a strictly stronger result than syntactic type soundness, and we have shown how to prove it at a higher level of abstraction than in prior work by exploiting the features of a modern separation logic, Iris. We conclude the paper by illustrating that our logical approach to type soundness is eminently scalable and practical. We do so by describing a general recipe for extending the logical approach to different languages, type systems, and program properties. Additionally, we offer a brief discussion of recent work that has employed the logical approach in practice, and provide references to papers and online tutorials that show how to mechanize logical type soundness proofs in Coq.

Applying the logical approach to other languages. We have studied the logical approach here in the context of the simple programming language **MyLang**, which exhibits a fairly pedestrian set of features. To apply the logical approach to a different language or a different type system, one roughly has to follow the following three steps:

1. Instantiate Iris with the language. The most common way to instantiate Iris with a programming language of choice is to start by defining its syntax and operational semantics. Iris's program logic (whose primary component is the connective for weakest preconditions) is language-generic. It is parametric in the type of expressions, values, states, and a reduction relation [Jung et al. 2018b, §7.3].

Instead of defining the syntax and operational semantics of a language from scratch, Iris's default language *HeapLang* could be reused. HeapLang is similar to **MyLang**, but comes with a number of additional features, such as arrays. Some programming languages are well-suited to be defined as a shallow embedding on top of HeapLang (see *e.g.,* Hinrichsen et al. [2021] for a language with message-passing primitives *à la* session types).

2. Define reasoning principles for the language. After having defined the syntax and operational semantics of the programming language, one needs to define logical connectives for ownership of physical resources (like the points-to connective $\ell \mapsto v$) and program reasoning (like the weakest precondition connective $\mathrm{wp}\ e\ \{\Phi\}$).

If the programming language fits into Iris's common format for small-step operational semantics, Iris's generic program logic and definition of weakest preconditions can be used. Additionally, if the language has a simple memory model like **MyLang**'s, Iris generic library for the points-to connective $\ell \mapsto v$ can be used. If the language has a more sophisticated memory model (*e.g.,* a block/offset-based memory model like CompCert's [Leroy and Blazy 2008] as used in RustBelt [Jung et al. 2018a, 2021; Jung 2020] and RefinedC [Sammler et al. 2021], or a weak memory model like iGPS's [Kaiser et al. 2017] as used in RustBelt Relaxed [Dang et al. 2020]), or if it has additional physical resources (*e.g.,* a program counter and registers as in a low-level capability machine [Georges et al. 2021]), custom connectives for ownership of physical resources need to be defined. Such connectives can be defined by combining existing libraries or by rolling one's own library using ghost state. This library can then be plugged into the generic weakest precondition connective provided by Iris's program logic.

Instead of reusing Iris's generic program logic (and its definition of weakest preconditions), Iris can also be used to define a custom program logic. That way, more flexibility can be obtained, while one can still reuse Iris's base logic—which comprises the usual connectives of the assertion language of separation logic (such as $*$ and $-\!\!*$) and basic modalities (such as $\triangleright$). This is useful, for example, to obtain a weakest precondition in big-step rather than small-step style (see *e.g.,* Timany et al. [2018]; Gregersen et al. [2021]), or to establish program properties that are out of scope of Iris's generic program logic (*e.g.,* security, see Frumin et al. [2021a]; Gregersen et al. [2021]).

Instead of building a custom program logic from scratch, one can also define custom reasoning principles in terms of Iris's generic weakest preconditions (see *e.g.,* Timany and Birkedal [2019] for a notion of context-local weakest precondition to reason about continuations).

3. Define ghost theories for modeling the type system. Once reasoning principles for the programming language have been set up, one needs to define suitable ghost theories for modeling the features of the type system. In this paper, we have seen three instances of ghost theories: impredicative invariants for modeling higher-order references (§6.8), ghost counters for monotonically increasing counters as used in the symbol ADT example (§7), and specification resources for proving representation independence (§8.3). Other examples of such ghost theories are RustBelt's lifetime logic for modeling Rust's lifetime and borrowing mechanism [Jung et al. 2018a, 2021; Jung 2020; Dang et al. 2020], and Actris's dependent separation protocols for modeling session types [Hinrichsen et al. 2020, 2022, 2021].

Ghost theories are defined using Iris's mechanism of *higher-order ghost state* [Jung et al. 2016]. This mechanism is based on PCMs (partial commutative monoids)—as found in many separation logics—but generalizes them with a step-indexed notion of equality. Iris's generalized PCMs are called *step-indexed resource algebras*, or *cameras* for short. Note that while we have presented impredicative invariants as a primitive of Iris, they are in fact defined in terms of Iris's higher-order ghost state mechanism. Some recent work, *e.g.,* Giarrusso et al. [2020], uses higher-order ghost state directly instead of invariants.

Recent work that employs the logical approach. Over the past several years, the logical approach in Iris has been deployed in a variety of applications. For instance, it has been used for a machine-checked proof of type soundness of a significant subset of the Rust programming language [Jung et al. 2018a, 2021; Jung 2020; Dang et al. 2020], an extension of Scala's core type system DOT [Giarrusso et al. 2020], session types [Hinrichsen et al. 2021], and refinement types for the C programming language [Sammler et al. 2021]. Aside from type soundness, it has also been used to prove robust safety [Swasey et al. 2017; Sammler et al. 2020], various forms of representation independence and program refinement [Krogh-Jespersen et al. 2017; Tassarotti et al. 2017; Timany et al. 2018; Timany and Birkedal 2019; Frumin et al. 2018, 2021b; Jacobs et al. 2021], and various security properties [Frumin et al. 2021a; Gregersen et al. 2021; Georges et al. 2021]. Instead of discussing these applications in detail, we highlight some interesting differences between our presentation of the logical approach and theirs.

In this paper we have considered a "standard" (unrestricted) type system, in which variables can be used any number of times and types are not used to enforce a discipline of resource ownership. However, since Iris is a separation logic, it is in fact designed to reason about ownership and is thus ideally suited to applying the logical approach to substructural type systems. For example, Jung et al. [2018a, 2021]; Jung [2020]; Dang et al. [2020] used Iris to model Rust's type system, which uses a strict ownership discipline to guarantee memory safety and data-race freedom in the context of low-level programming paradigms, and Tassarotti et al. [2017]; Hinrichsen et al. [2021] used Iris to model session types, which use ownership to enforce protocols compliance in message-passing communication.

The crucial difference between an unrestricted and substructural type system is whether to make the value interpretation $[\![A]\!]$ persistent or not. In an unrestricted type system (such as the type system for **MyLang** in this paper), the value interpretation $[\![A]\!]$ is persistent for *any* type A. In a substructural type system (such as Rust or session types) the value interpretation $[\![A]\!]$ is *not* persistent for types that denote ownership (such as mutable references and channels), while it is persistent for "copyable" types (such as integers, Booleans and shared references).

With regard to refinement proofs, let us point out two differences from some of the above-cited papers. First, in this paper we had to unfold the logical refinement judgment and carry out a proof in terms of its definition in Iris (see §8.6 for how this is done for the example of concurrent stacks). In contrast, Frumin et al. [2018, 2021b] presented a *logic* for doing such refinement proofs at a higher level of abstraction, and showed how it simplifies reasoning about refinements.

Second, in this paper, we have used Iris to prove termination-*insensitive* program refinement, in which any non-terminating program is a contextual refinement of any other program. In contrast, Tassarotti et al. [2017] developed a version of Iris to prove termination-*preserving* refinements. While their approach establishes a stronger version of refinement, it also has some limitations—it can only be used in the context of languages with countable non-determinism, and for refinement proofs that involve finite stuttering. Recent work by Spies et al. [2021] overcomes these limitations in a non-concurrent setting by employing a *transfinite* version of step-indexing, where steps are modeled using ordinals instead of natural numbers. An interesting direction for future work is to scale this approach to the concurrent setting.

Coq material. To develop Iris proofs in practice, nearly all Iris users make use of the Iris Proof Mode [Krebbers et al. 2017b, 2018], which provides tactics and other infrastructure for carrying out separation logic proofs in Coq. While a presentation of the Iris Proof Mode is beyond the scope of this paper, we provide some references to online tutorials:

- https://gitlab.mpi-sws.org/iris/tutorial-popl20/
  This tutorial (which was presented at the POPL'20 conference) demonstrates how to prove

logical type soundness in Coq. The structural of this tutorial largely follows §5–§7, but uses Iris's default language HeapLang (and the infrastructure that Iris provides for HeapLang) instead of the language **MyLang**. This tutorial comes with exercises.

- https://gitlab.mpi-sws.org/iris/tutorial-popl21/
  This tutorial (which was presented at the POPL'21 conference, and is based on an earlier version at POPL'18) does not specifically target logical type soundness, but provides an introduction to reasoning about concurrent programs in Iris. This tutorial comes with exercises.
- https://github.com/tchajed/iris-simp-lang/
  This tutorial demonstrates how to instantiate Iris with a custom language, based on a stripped-down version of HeapLang.
- https://gitlab.mpi-sws.org/iris/examples (directory logrel/F_mu_ref_conc)
  This development contains a mechanization of the semantic type soundness proof (§5–§7) and representation independence proof (§8) of this paper. Rather than reusing Iris's language HeapLang (and the infrastructure that Iris provides for HeapLang), it instantiates Iris with a custom language that is closer to **MyLang**.
- https://gitlab.mpi-sws.org/FP/semantics-course/
  This development accompanies lecture notes from a course on Semantics taught periodically at Saarland University (https://plv.mpi-sws.org/semantics-course/lecturenotes.pdf). The first half of the notes cover semantic type soundness and logical relations, formalized directly in the traditional, explicitly step-indexed style; the second half of the notes offer a tutorial on Iris, with the ulterior motive of showing how to re-implement the semantic models of the first half in the logical style. The language considered is similar to **MyLang**.

ACKNOWLEDGMENTS

We wish to thank our many collaborators on the Iris project for helpful discussions. This research was supported in part by the Dutch Research Council (NWO), project 016.Veni.192.259, in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289), and in part by generous gifts from Google. Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project.

REFERENCES

Martín Abadi and Gordon D. Plotkin. 1990. A PER model of polymorphism and recursive types. In *LICS*. 355–365. https://doi.org/10.1109/LICS.1990.113761

Amal Ahmed. 2004. *Semantics of types for mutable state*. Ph. D. Dissertation. Princeton University.

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A stratified semantics of general references. In *LICS*. 75–86. https://doi.org/10.1109/LICS.2002.1029818

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. 340–353. https://doi.org/10.1145/1480881.1480925 Technical appendix: http://www.ccs.neu.edu/home/amal/papers/sdri/main-long.pdf.

Amal J. Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP (LNCS, Vol. 3924)*. 69–83. https://doi.org/10.1007/11693024_6 Technical appendix: http://www.ccs.neu.edu/home/amal/papers/lr-recquant-techrpt.pdf.

Stuart Allen. 1987. *A non-type-theoretic semantics for type-theoretic language*. Ph. D. Dissertation. Cornell University.

Andrew W. Appel. 2001. Foundational proof-carrying code. In *LICS*. 247–256. https://doi.org/10.1109/LICS.2001.932501

Andrew W. Appel and Amy P. Felty. 2000. A Semantic Model of Types and Machine Instuctions for Proof-Carrying Code. In *POPL*. 243–253. https://doi.org/10.1145/325694.325727

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122. https://doi.org/10.1145/1190216.1190235

E. S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. 1990. Functorial polymorphism. *TCS* 70, 1 (1990), 35–64. https://doi.org/10.1016/0304-3975(90)90151-7

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *TOPLAS* 38, 4 (2016), 14:1–14:94. https://doi.org/10.1145/2837022

Nick Benton. 2006. Abstracting allocation: The new new thing. In *CSL*. 182–196. https://doi.org/10.1007/11874683_12

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *ICFP*. 97–108. https://doi.org/10.1145/1596550.1596567

Nick Benton and Nicolas Tabareau. 2009. Compiling functional types to relational specifications for low level imperative code. In *TLDI*. 3–14. https://doi.org/10.1145/1481861.1481864

Nick Benton and Uri Zarfaty. 2007. Formalizing and verifying semantic type soundness of a simple compiler. In *PPDP*. 1–12. https://doi.org/10.1145/1273920.1273922

Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. 2013. Step-indexed relational reasoning for countable nondeterminism. *LMCS* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:4)2013

Lars Birkedal and Robert Harper. 1999. Relational interpretations of recursive types in an operational setting. *Information and Computation* 155, 1-2 (1999), 3–63. https://doi.org/10.1006/inco.1999.2828

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132. https://doi.org/10.1145/1926385.1926401

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A concurrent logical relation. In *CSL (LIPIcs, Vol. 16)*. 107–121. https://doi.org/10.4230/LIPIcs.CSL.2012.107

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. Realisability semantics of parametric polymorphism, general references and recursive types. *MSCS* 20, 4 (2010), 655–703. https://doi.org/10.1017/S0960129510000162

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*. 259–270. https://doi.org/10.1145/1040305.1040327

Stephen Brookes. 2007. A semantics for concurrent separation logic. *TCS* 375, 1-3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Kim B. Bruce and John C. Mitchell. 1992. PER models of subtyping, recursive types and higher-order polymorphism. In *POPL*. 316–327. https://doi.org/10.1145/143165.143230

Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

Karl Crary and Robert Harper. 2007. Syntactic logical relations for polymorphic and recursive types. *ENTCS* 172 (2007), 259–299. https://doi.org/10.1016/j.entcs.2007.02.010

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *PACMPL* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about object capabilities with logical relations and effect parametricity. In *EuroS&P*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (LNCS, Vol. 6183)*. 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Derek Dreyer. 2018. Milner Award Lecture: The type soundness theorem that you really want to prove (and now you can). Keynote talk at POPL 2018, https://www.youtube.com/watch?v=8Xyk_dGcAwk&ab_channel=POPL2018.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical step-indexed logical relations. In *LICS*. 71–80. https://doi.org/10.1109/LICS.2009.34

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical step-indexed logical relations. *LMCS* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *JFP* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X Technical appendix: http://www.mpi-sws.org/tr/2012-001.pdf.

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. 185–198. https://doi.org/10.1145/1706299.1706323

Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey, and Jan Menz. 2022. Semantics of type systems (Lecture notes). Available at https://plv.mpi-sws.org/semantics-course/.

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *TCS* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021a. Compositional non-interference for fine-grained concurrent programs. In *S&P*. 1416–1433. https://doi.org/10.1109/SP40001.2021.00003

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021b. ReLoC Reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *LMCS* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR (LNCS, Vol. 6269)*. 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *PACMPL* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris. *PACMPL* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

Jean-Yves Girard. 1972. *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur.* Ph. D. Dissertation. Universite Paris VII.

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *PACMPL* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434291

Dan Grossman, Greg Morrisett, and Steve Zdancewic. 2000. Syntactic type abstraction. *TOPLAS* 22, 6 (2000), 1037–1080. https://doi.org/10.1145/371880.371887

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.).* Cambridge University Press. https://www.cs.cmu.edu/~rwh/pfpl/index.html

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-type based reasoning in separation logic. *PACMPL* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous session-type based reasoning in separation logic. *LMCS* 18, 2 (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. (2021), 178–198. https://doi.org/10.1145/3437992.3439914

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *POPL*. 133–146. https://doi.org/10.1145/1926385.1926402

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. 59–72. https://doi.org/10.1145/2103656.2103666

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully abstract from static to gradual. *PACMPL* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434288

Achim Jung and Jerzy Tiuryn. 1993. A new characterization of lambda definability. In *TLCA (LNCS, Vol. 664)*. 245–257. https://doi.org/10.1007/BFb0037110

Ralf Jung. 2020. *Understanding and evolving the Rust programming language.* Ph. D. Dissertation. Saarland University. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *CACM* 64, 4 (2021), 144–152. https://doi.org/10.1145/3418295

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: Prophecy variables in separation logic. *PACMPL* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

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*. 637–650. https://doi.org/10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: reasoning about release-acquire consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*. 141–152. https://doi.org/10.1145/1111037.1111050

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3093333.3009855

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric semantics of reactive programs. In *LICS*. 257–266. https://doi.org/10.1109/LICS.2011.38

Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially substructural types. In *ICFP*. 41–54. https://doi.org/10.1145/2364527.2364536

Jean-Louis Krivine. 1994. Classical logic, storage operators and second-order lambda-calculus. *APAL* 68, 1 (1994), 53–78. https://doi.org/10.1016/0168-0072(94)90047-7

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. 218–231. https://doi.org/10.1145/3093333.3009877

Søren B. Lassen and Paul Blain Levy. 2007. Typed normal form bisimulation. In *CSL (LNCS, Vol. 4646)*. 283–297. https://doi.org/10.1007/978-3-540-74915-8_23

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Xavier Leroy and Pierre Weis. 1991. Polymorphic type inference and assignment. In *POPL*. 291–302. https://doi.org/10.1145/99583.99622

David B. MacQueen. 1984. Modules for Standard ML. In *LFP*. 198–207. https://doi.org/10.1145/800055.802036

David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. 1986. An ideal model for recursive polymorphic types. *Information and Control* 71, 1/2 (1986), 95–130. https://doi.org/10.1016/S0019-9958(86)80019-5

Robin Milner. 1978. A theory of type polymorphism in programming. *JCSS* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

John C. Mitchell. 1986. Representation independence and data abstraction. In *POPL*. 263–276. https://doi.org/10.1145/512644.512669

John C. Mitchell and Gordon D. Plotkin. 1988. Abstract types have existential type. *TOPLAS* 10, 3 (1988), 470–502. https://doi.org/10.1145/44501.45065

Andrzej S. Murawski and Nikos Tzevelekos. 2019. Higher-order linearisability. *J. Log. Algebraic Methods Program.* 104 (2019), 86–116. https://doi.org/10.1016/j.jlamp.2019.01.002

Emeric Nasi. 2011. Modify any Java class field using reflection. Website: https://blog.sevagas.com/Modify-any-Java-class-field-using-reflection.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric parametricity. In *ICFP*. 135–148. https://doi.org/10.1145/1596550.1596572

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ICFP*. 166–178. https://doi.org/10.1145/2784731.2784764

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *CSL (LNCS, Vol. 2142)*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn and Robert D. Tennent. 1992. Semantics of local variables. In *Applications of Categories in Computer Science*. London Mathematical Society Lecture Note Series, Vol. 177. 217–238.

Benjamin C. Pierce. 2002. *Types And Programming Languages*. MIT Press.

Andrew M. Pitts. 1996. Relational properties of domains. *Information and Computation* 127, 2 (1996), 66–90. https://doi.org/10.1006/inco.1996.0052

Andrew M. Pitts. 2005. Typed operational reasoning. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce (Ed.). The MIT Press, Chapter 7, 245–289.

Andrew M. Pitts and Ian Stark. 1998. Operational reasoning for functions with local state. In *HOOTS*.

Gordon D. Plotkin and Martín Abadi. 1993. A logic for parametric polymorphism. In *TLCA (LNCS, Vol. 664)*. 361–375. https://doi.org/10.1007/BFb0037118

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris (LNCS, Vol. 19)*. 408–423. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83*. 513–523.

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *JFP* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *PACMPL* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

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*. 158–174. https://doi.org/10.1145/3453483.3454036

Jan Schwinghammer, Lars Birkedal, François Pottier, Bernhard Reus, Kristian Støvring, and Hongseok Yang. 2013. A step-indexed Kripke model of hidden state. *MSCS* 23, 1 (2013), 1–54. https://doi.org/10.1017/S0960129512000035

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*. 80–95. https://doi.org/10.1145/3453483.3454031

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: Resourceful reasoning for the later modality. *PACMPL* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

Kristian Støvring and Søren B. Lassen. 2007. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*. 161–172. https://doi.org/10.1145/1190216.1190244

Eijiro Sumii. 2009. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL (LNCS, Vol. 5771)*. 455–469. https://doi.org/10.1007/978-3-642-04027-6_33

Eijiro Sumii. 2010. A bisimulation-like proof method for contextual properties in untyped λ-calculus with references and deallocation. *TCS* 411, 51-52 (2010), 4358–4378. https://doi.org/10.1016/j.tcs.2010.09.009

Eijiro Sumii and Benjamin C. Pierce. 2007. A bisimulation for type abstraction and recursion. *JACM* 54, 5 (2007), 26. https://doi.org/10.1145/1284320.1284325

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular reasoning about separation of concurrent data structures. In *ESOP (LNCS, Vol. 7792)*. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *PACMPL* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The type and effect discipline. *Information and Computation* 111, 2 (1994), 245–296. https://doi.org/10.1006/inco.1994.1046

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Jacob Thamsborg and Lars Birkedal. 2011. A Kripke logical relation for effect-based program transformations. In *ICFP*. 445–456. https://doi.org/10.1145/2034773.2034831

Amin Timany. 2018. *Contributions in programming languages theory.* Ph. D. Dissertation. KU Leuven. https://lirias.kuleuven.be/retrieve/510052

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *PACMPL* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Mads Tofte. 1990. Type inference for polymorphic references. *Inf. Comput.* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013a. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600

Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013b. Logical relations for fine-grained concurrency. In *POPL*. 343–356. https://doi.org/10.1145/2429069.2429111 Technical appendix: https://people.mpi-sws.org/~dreyer/papers/relcon/appendix.pdf.

Andrew K. Wright. 1995. Simple imperative polymorphism. *Lisp Symb. Comput.* 8, 4 (1995), 343–355. https://doi.org/10.1007/BF01018828

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Hongseok Yang. 2007. Relational separation logic. *TCS* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036