Mechanized Logical Relations for Termination-Insensitive Noninterference

SIMON ODDERSHEDE GREGERSEN, Aarhus University, Denmark
JOHAN BAY, Aarhus University, Demark
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present an expressive information-flow control type system with recursive types, existential types, label polymorphism, and impredicative type polymorphism for a higher-order programming language with higher-order state. We give a novel semantic model of this type system and show that well-typed programs satisfy termination-insensitive noninterference. Our semantic approach supports compositional integration of syntactically well-typed and syntactically ill-typed—but semantically sound—components, which we demonstrate through several interesting examples. We define our model using logical relations on top of the Iris program logic framework; to capture termination-insensitivity, we develop a novel language-agnostic theory of Modal Weakest Preconditions. We formalize all of our theory and examples in the Coq proof assistant.

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Formal security models**; • **Theory of computation** → *Separation logic*; **Program verification**.

Additional Key Words and Phrases: Logical Relations, Information-Flow Control, Program Logics, Iris

ACM Reference Format:
Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized Logical Relations for Termination-Insensitive Noninterference. *Proc. ACM Program. Lang.* 5, POPL, Article 10 (January 2021), 29 pages. https://doi.org/10.1145/3434291

1 INTRODUCTION

Systems for information-flow control put restrictions on how a program's outputs are related to its inputs. Such systems establish various notions of *noninterference* [Goguen and Meseguer 1982], conveying that observable aspects of the program's behavior is independent of its sensitive inputs. Information-flow control enforcement is often specified as a static type system (*e.g.*, Abadi et al. [1999]; Arden and Myers [2016]; Heintze and Riecke [1998]; Lourenço and Caires [2015]; Myers [1999]; Simonet [2003b]) or via an encoding into an existing type system (*e.g.*, Algehed and Russo [2017]; Gregersen et al. [2019]; Li and Zdancewic [2006]; Pottier and Simonet [2003]; Russo [2015]; Russo et al. [2008]; Vassena et al. [2018]). Modern programming languages have rich type systems featuring, *e.g.*, higher types, reference types, and abstract types, which are all essential for modern software engineering practice and for implementing reusable software components. Naturally, modern practical information-flow secure languages have to meet the same demands, but as the complexity of the type system increases, so does the burden of proving the type system sound.

Authors' addresses: Simon Oddershede Gregersen, Aarhus University, Denmark, gregersen@cs.au.dk; Johan Bay, Aarhus University, Demark, bay@cs.au.dk; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2021/1-ART10
https://doi.org/10.1145/3434291

In this paper, we prove soundness of an expressive information-flow control type system for a higher-order language with higher-order state. The type system is an extension of the fine-grained type system of Rajani and Garg [2020] and the type system of Flow Caml [Simonet 2003a], with recursive types, existential types, and impredicative type polymorphism (in addition to existing reference types and function types). The main high-level goal of our work is to prove that the type system satisfies *termination-insensitive noninterference* using a semantic model. Since such type soundness results for expressive type systems involve myriads of details (as exhibited by a 100 pp. chapter in a technical appendix [Rajani and Garg 2020]), we formalize our model in a proof assistant and use it to give a full mechanization of all our technical results.

Even with a very expressive type system, any static type system is necessarily overly conservative. This entails that there is a large body of programs that cannot be type-checked while still being information-flow secure for reasons too subtle for the type system to verify. This includes, *e.g.*, low-level implementations of data structures that are optimized for efficiency and systems governed by security policies that rely on value-dependency or dynamic run-time information. Our *semantic* approach to establishing noninterference enables compositional integration of syntactically well-typed components with syntactically ill-typed but semantically sound components: only the syntactically ill-typed parts need to be carefully verified to show that the entire program enjoys the security property.

To meet our goals, we define a novel logical-relations model of our proposed type system. We define our logical-relations model in the Iris separation logic framework [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a]. We do this to (1) define and reason about our logical-relations model at a high level of abstraction, (2) side-step the well-known problem of *type-world circularity*[1] [Ahmed 2004; Ahmed et al. 2002; Birkedal et al. 2011] when defining logical-relations models of programming languages with higher-order state in the presence of impredicative polymorphism, and (3) to leverage the Coq formalization and the MoSeL framework [Krebbers et al. 2018] to fully mechanize all examples and technical results.

Challenges. Extending the earlier type system is mostly straightforward: similarly to how ordinary functions in languages with effects may have latent effects, polymorphic functions may also have latent effects and thus they must be annotated with a label expressing a lower bound on these effects.

So what is new and challenging about our semantic model? In summary, we address three major challenges: (1) combining unary and binary logical-relations models in the presence of impredicative polymorphism, and (2) constructing "logical" [Dreyer et al. 2009] logical-relations models for termination-insensitive reasoning while (3) soundly allowing syntactically ill-typed but semantically secure programs to be composed with syntactically well-typed programs. We now explain each of these points.

To construct a logical-relations model of a termination-insensitive information-flow control type system in the presence of state, it is necessary to combine both a unary and a binary model; when branching on high-labeled information, it is crucial that that the two branches, independently, do not modify low-labeled references, to avoid implicit leaks through the store. This is commonly know as the *confinement lemma* in proofs of noninterference.

[1]To ensure that heap updates are type-preserving, the model of mutable references types $\mathsf{ref}(\tau)$ needs to keep track of the semantics of τ; this is usally done using a store typing (a World) Θ mapping locations to types. The model of $\mathsf{ref}(\tau)$ then needs to refer to Θ to checker whether a location maps to the appropriate type. This in turn means the model of all types has to take Θ as an argument, introducing what is known as the type-world circularity—this kind of recursive domain equation has no solution in the category of sets.

When developing logical-relations models for languages with state in the presence of impredicative polymorphism, one needs to work with so-called step-indexed recursive Kripke worlds which are used to describe the semantics of the contents of the heap [Ahmed 2004; Birkedal et al. 2011]. These step-indexed Kripke worlds imply that both the binary and the unary logical relations have to be step-indexed. Binary logical relations usually tie the logical steps of the recursive Kripke worlds to the physical steps taken by only one of the two programs in the relation. However, this causes a mismatch in the number of steps when we want to combine the individual unary logical relatedness of two programs to conclude that they are in the binary relation. To solve this problem, one novelty of our binary logical-relations model is that we count the steps taken by the programs on *both sides* of the relation. Rajani and Garg [2020] circumvent this problem by using syntactic worlds which does not scale to impredicative polymorphism. This also means that their logical relations are defined over syntactically well-typed programs and hence cannot be used for reasoning about syntactically ill-typed but semantically well-typed programs as we do in this paper.

The idea of using a more expressive logic to simplify the definition of logical-relations models is not novel and goes back to Plotkin and Abadi [1993] who used second order logic for modeling System F and Dreyer et al. [2009] who used a logic with step-indexing to model recursive types. It has since been used for defining logical relations models for a variety of programming languages and features, *e.g.*, an ML-style language with concurrency [Krebbers et al. 2017b], a Haskell style ST monad [Timany et al. 2018], a concurrent ML-style language featuring continuations [Timany and Birkedal 2019], and the Rust programming language [Jung et al. 2018a]. All these models are either unary logical-relations models used for proving type safety or binary logical-relations models for proving traditional contextual program refinement. Intuitively e contextually refinenes e' if whenever e terminates with some value v, then e' must also terminate with some value v' and then v and v' should be suitably related. In symbols:

$$e \Downarrow v \Rightarrow e' \Downarrow v' \wedge v \approx v'.$$

This is crucially different from the idea of *termination-insensitive* noninterference where two programs are equivalent if, assuming that *both* e and e' terminate, then their resulting values should be suitably related:

$$e \Downarrow v \wedge e' \Downarrow v' \Rightarrow v \approx v'.$$

The termination-insensitive nature of the equivalence is the reason why the approaches taken heretofore on expressing logical-relations models in program logics cannot be extended to support reasoning about termination-insensitive noninterference. Moreover, these works do not consider logical-relations models that incorporate both a unary and a binary relation.

The core challenge here is to properly hide the details of step-indexing and recursive Kripke worlds. To this end, the base logic of Iris provides modalities to reason about step-indices and ghost resources (logical counterparts of recursive Kripke worlds). Yet, using these logical facilities directly, while hiding a lot of details, still requires us to think and work in terms of step-indices and explicit resource updates (manipulating ghost resources). Previous work [Krebbers et al. 2017b; Timany and Birkedal 2019; Turon et al. 2013], addressed this problem by defining the logical relation models using Iris' weakest precondition predicates, which themselves are defined using logical step-indexing and ghost resource modalities but which, importantly, come with high-level reasoning principles that hide those details. Iris' weakest precondition predicates were a good match for contextual refinement: we can express "if e terminates then so does e'" as a weakest precondition for e where the post condition states that e' terminates, *i.e.*, $\mathsf{wp}\ e\ \{v.\ e' \Downarrow v \wedge v = v'\}$. As discussed above, this is crucially different from termination-insensitive noninterference: "if both programs terminate then ...". This prevents us from using weakest preconditions to model our logical relations. One might be tempted to consider nested weakest preconditions: $\mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ e'\ \{v'.\ v = v'\}\}$. This

formulation does indeed imply that if both programs terminate then their results are equal, however, this formulation is too strong and in particular makes it impossible to employ the kind of modular reasoning that is essential to proving the fundamental theorem of logical relations. Intuitively, this is because such a formulation requires us to reason about the execution of e' *after* the completion of execution of e. Technically, this formulation does not admit the so-called binary bind rule, *cf.* Lemma 3.6.

In place of weakest preconditions we introduce and use a novel program logic construct that we call *Modal Weakest Preconditions* (MWP). Our Modal Weakest Precondition theory is language agnostic, parameterized by a modal operator, and general enough to allows us to define both a unary and a binary predicate for reasoning about computations using the same theory. Indeed, the generality is one of the key strengths of our theory. Different instantiations automatically inherit a set of basic structural proof rules that hold irrespective of the particular modality and programming language. For particular instantiations, one can then prove more specific proof rules, *e.g.*, for heap-manipulating operations and for how the instantiation interacts with other instantiations with different modal operators. We use three different instantiations for our logical-relations model and two more for concrete examples. The generality of our MWP theory allows us to define our binary logical relations model to be weak enough so as to allow us to reason modularly as discussed above. Yet, the interaction between different instantiations of MWP's (which is proven generally and not particularly for our programming language) allows us to strengthen this definition, in order to combine unary and binary logical relations (see Lemma 3.7) and to prove certain examples that require stronger reasoning principles (see §5).

Another challenge worth noting is the modeling of reference types. Intuitively, two values are related at the reference type $\mathsf{ref}(\tau)$ if they are both locations that invariantly store values that are related at type τ. Previous work used Iris invariants to formalize this idea. In our case, we can only use Iris invariants for our binary logical relation; for the unary logical relation we need to use a more refined approach, see the discussion in Section 3.

Contributions. In summary, we make the following contributions:

- We present the first logical-relations model of an information-flow type system with recursive types, existential types, and impredicative polymorphism for a language with higher-order state. To the best of our knowledge, this is also the first soundness proof of such an expressive information-flow type system irrespective of method.
- We present the first "logical" logical-relations model that incorporates both a unary and a binary relation and termination-insensitive reasoning.
- We introduce a new theory of *Modal Weakest Preconditions* (MWP) that allows us to construct novel logical-relations models for proving relational properties of programs that were out of reach of existing techniques.
- We propose a methodology that allows us to establish *termination-insensitive noninterference* of syntactically ill-typed but semantically secure programs while allowing these programs to be composed with syntactically well-typed programs and showcase multiple examples.
- We also show that our logical-relations model allows us to prove "*free theorems*" for our information-flow control type system.
- We formalize all of the theory and examples on top of the Iris program logic framework in the Coq proof assistant using the MoSeL framework [Krebbers et al. 2018]. The Coq formalization can be found online at

https://github.com/logsem/iris-tini.

2 THE λ_{sec} LANGUAGE

We present the syntax and operational semantics of the subject of our study: a higher-order functional call-by-value language with higher-order state which we equip with an information-flow control type system with recursive types, existential types, label polymorphism, and impredicative type polymorphism.

2.1 Syntax and Semantics

The syntax of λ_{sec} is as follows:

$$
\begin{aligned}
\circledcirc &::= + \mid - \mid * \mid = \mid < \\
e \in \mathit{Expr} &::= x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid n \in \mathbb{N} \mid n \circledcirc n \mid \lambda x.\, e \mid e\ e \mid \Lambda\, e \mid \Lambda\!\!\Lambda\, e \mid e\ \_ \mid \\
&\quad \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e, e) \mid \pi_i\, e \mid \mathsf{inj}_i\, e \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inj}_i\ \Rightarrow e_i\ \mathsf{end} \\
&\quad \mid \mathsf{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e \\
v \in \mathit{Val} &::= () \mid \mathsf{true} \mid \mathsf{false} \mid n \in \mathbb{N} \mid \iota \in \mathit{Loc} \mid \lambda x.\, e \mid \Lambda\, e \mid \Lambda\!\!\Lambda\, e \mid \mathsf{fold}\ v \mid \mathsf{pack}\ v \mid (v, v) \mid \mathsf{inj}_i\, v \\
\ell &::= \kappa \mid l \in \mathcal{L} \mid \ell \sqcup \ell \\
\tau \in \mathit{LType} &::= t^{\ell} \\
t \in \mathit{Type} &::= \alpha \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \tau \times \tau \mid \tau + \tau \mid \tau \xrightarrow{\ell} \tau \mid \forall_{\ell}\, \alpha.\, \tau \mid \forall_{\ell}\, \kappa.\, \tau \mid \exists \alpha.\, \tau \mid \mu\, \alpha.\, \tau \mid \mathsf{ref}(\tau)
\end{aligned}
$$

The term language is mostly standard but note that there are no types in terms; we write $\Lambda\, e$ for (unlabeled) type abstraction and $e\ \_$ for type application. Similarly, we write $\Lambda\!\!\Lambda\, e$ for label abstraction and $e\ \_$ for label application. $\mathsf{fold}\ e$ and $\mathsf{unfold}\ e$ are the special term constructs for iso-recursive types. $\mathsf{ref}(e)$ allocates a new reference, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We introduce syntactic sugar for let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ defined as $(\lambda x.\, e_2)(e_1)$, and sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

The set of types of λ_{sec} is parameterized over an arbitrary bounded join-semilattice $\mathcal{L}$ with ordering $\sqsubseteq$. The lattice ordering $\sqsubseteq$ defines the security policy: if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \not\sqsubseteq \ell_1$ then information with label ℓ_1 may influence information with label ℓ_2 but not the other way around. We write $\bot$ for the least element. Syntactically, a label ℓ is either a label variable κ, a label l drawn from the lattice $\mathcal{L}$, or the formal least upper bound (join) of two labels.

Types are syntactically either labeled or unlabeled; we use τ to range over labeled types and t to range over unlabeled types. As standard, a term of a labeled type t^{ℓ}, is a term of the (unlabeled) type t labeled with the security label ℓ. Note that type abstraction, existential types, and recursive types abstract over unlabeled types.

The unlabeled types of λ_{sec} include basic types such as the unit type, Booleans, natural numbers, products, and sums. The function type $\tau \xrightarrow{\ell} \tau$ is annotated with a label ℓ. This label, which we refer to as a *latent effect label*, denotes a *lower bound* on the write effects of the function body. The type system will ensure that any reference that the function may write to has a label that is ℓ or higher according to the lattice ordering. This is necessary to prevent implicit information leaks through the store where programs have write effects that conditionally depend on sensitive information. Let ι be a reference with contents of type $\mathbb{N}^{\bot}$ and h a variable of type $\mathbb{B}^{\top}$ with $\top \not\sqsubseteq \bot$. When control flow depends on h invoking a function like $f \triangleq \lambda\_.\, \iota \leftarrow 1$ implicitly leaks h through the store, *e.g.*,

$$\mathsf{if}\ h\ \mathsf{then}\ f\ ()\ \mathsf{else}\ ()$$

by subsequently observing whether the write effects happened or not. The label $\bot$ is a lower bound of the side-effects of f and it may not be invoked when control flow depends on h with label $\top$ as

$\top \not\sqsubseteq \bot$. For the same reason, type-polymorphic types $\forall_\ell\, \alpha.\, \tau$ and label-polymorphic types $\forall_\ell\, \kappa.\, \tau$ also include a label annotation ℓ. We also refer to those as latent effect labels. Finally, types also include existential types $\exists \alpha.\, \tau$, recursive types $\mu\, \alpha.\, \tau$, and the type $\mathsf{ref}(\tau)$ of memory locations storing values of type τ.

The states σ of λ_{sec} are modeled as finite partial functions from locations to values. We define a small-step operational semantics $(\sigma, e) \to (\sigma', e')$ of λ_{sec} using left-to-right call-by-value evaluation contexts. These definitions are entirely standard and can be found in the appendix.

2.2 Information-Flow Control Type System

The type system of λ_{sec} is very similar to the fine-grained type system of Rajani and Garg [2020] and the type system of Flow Caml [Simonet 2003a] but extended with recursive types, existential types, and impredicative polymorphic types. We write $\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau$ for the syntactic typing judgment which expresses that expression e has type τ under typing contexts Γ, Ξ, and Ψ. A typing context Γ maps free variables that may appear in e to their types. The type-level contexts Ξ and Ψ are sets of free type and label variables, respectively, that may appear in τ and Γ. The annotation pc is a label, often called the *program counter* label, denoting a lower bound on the write effects of e, *cf.*, how function types and the polymorphic types are annotated with a latent effect labels.

The typing relation is shown in Figure 1 and we discuss some of the important rules below. The syntactic label ordering relation $\Psi \vdash \ell_1 \sqsubseteq \ell_2$ is straightforward and relegated to the appendix. The *protected-at* relation $\tau \searrow \ell$ is defined as $t^{\ell'} \searrow \ell \triangleq \ell \sqsubseteq \ell'$, meaning that the label of the type is at least as high as ℓ.

When applying a function expression e_1 of type $\left(\tau_1 \xrightarrow{\ell_e} \tau_2\right)^{\ell}$ to an argument e_2 of type τ_1 the rule for function application (T-APP) requires that the program counter label pc is lower than the latent effect label ℓ_e to avoid implicit leaks through the store. In addition, the label ℓ of the function value must be below ℓ_e and the return type τ_2 must be protected at ℓ in order to prevent implicit leaks arising from the identity of the function that e_1 evaluates to: If not, then, for example, given ι is a reference of type $\mathsf{ref}(\mathbb{N}^\bot)$ and h a variable of type $\mathbb{B}^\top$ both programs (1) and (2) would be typeable at $\mathbb{N}^\bot$ while both leaking h.

$$\mathsf{let}\, f = \mathsf{if}\, h\, \mathsf{then}\, \lambda\_.\, 1\, \mathsf{else}\, \lambda\_.\, 0\, \mathsf{in}\, f\ () \tag{1}$$

$$\mathsf{let}\, f = \mathsf{if}\, h\, \mathsf{then}\, \lambda\_.\, \iota \leftarrow 1\, \mathsf{else}\, \lambda\_.\, \iota \leftarrow 0\, \mathsf{in}\, (f\ (); !\, \iota) \tag{2}$$

Our type system correctly handles these situations in two different ways: in (1) the leak is captured in the output type of f, and in (2) it is captured in the label of f itself. Indeed, according to our typing rules, since the identity of the function f in (1) depends on h, the type of the output of f will have a label that is at least $\top$. On the other hand, in (2), the latent effect label for f is $\bot$ but the label of the function value itself must be $\top$ as it depends on h and the function may not be invoked. Similar considerations apply to the rules T-TAPP and T-LAPP for type and label application.

The rules for case analysis (T-MATCH and T-IF) demand that both branches are typed with program counter label $pc \sqcup \ell$ to account for the fact that control flow depends on the information with label ℓ of the expression e being cased on. This ensures that the branches do not have write effects below ℓ, which would otherwise be dependent on more sensitive information. Similarly, the result type τ has to be protected at ℓ.

The rule for assignment (T-STORE) captures that the pc label acts as an effect lower bound. It requires that when assigning an expression of type τ to a reference of type $\mathsf{ref}(\tau)^\ell$, then the label of τ is protected at both pc and ℓ. The former enforces pc as a lower bound on effects and the latter prevents implicit leaks arising from the identity of the reference. If not, then, for example, given ι_1 and ι_2 are references of type $\mathsf{ref}(\mathbb{N}^\bot)$ and h a variable of type $\mathbb{B}^\top$ the program (3) would be typeable

at $\mathbb{N}^{\bot}$ while leaking h.

$$\iota_1 \leftarrow 0; \iota_2 \leftarrow 0; \mathsf{let}\, r = \mathsf{if}\, h\, \mathsf{then}\, \iota_1\, \mathsf{else}\, \iota_2\, \mathsf{in}\, r \leftarrow 1; !\,\iota_1 \tag{3}$$

Note that all expressions typed with an introduction rule gets a type with label $\bot$. Intuitively, introducing, *e.g.*, a pair with components τ_1 and τ_2 has no observable effect nor does it leak any information, and the label of τ_i already captures the information that may have influenced the component. The label can, however, freely be raised using T-SUB and the subtyping relation in Figure 2. The rule S-LABELED allows a term with label ℓ_1 to be treated as a term with label ℓ_2 if $\ell_1 \sqsubseteq \ell_2$; the rest of the subtyping rules are standard. Notice that T-SUB also allows the *pc* label to be freely weakened.

3 SEMANTIC MODEL

In this section we define our semantic model of λ_{sec}'s type system. The model formalizes a notion of *observer-sensitive equivalence* which only relates computations *from the perspective of some observer*. Concretely, the observer is modelled by a fixed but arbitrary label ζ drawn from the lattice $\mathcal{L}$. The intuition is that terms typed with a label higher than ζ are indistinguishable to the observer whereas terms typed with a label lower than ζ are *not*.

Our semantic model captures all invariants necessary to prove that the type system guarantees that well-typed programs satisfy noninterference (Theorem 4.4). In §5 we demonstrate that our model can also be used to prove that syntactically ill-typed programs are semantically secure—this allows us to safely compose syntactically ill-typed but semantically secure programs with syntactically well-typed programs. In §5.5, we show that the model can also be used to prove "free" theorems.

A central idea in the model is to interpret each type both as a binary relation (Figure 4) and as unary relation (Figure 6). The binary relation relates expressions that are observationally equivalent to a ζ observer, and the unary relation relates expressions that do not have any ζ-observable side-effects. The unary relation is used within the binary relation to relate terms *independently* when the label of the type is higher than the observer—such terms are indistinguishable to the observer as long as they do not have any visible side-effects.

In this section, we will show step-by-step how to define our model in Iris. The syntax of Iris is shown in Figure 3. Iris is a higher-order separation logic with propositions of type *iProp* and some custom connectives that we will explain as we go along.

We start by defining the binary and unary value relations (§3.1) followed by a brief intermezzo, where we present the Modal Weakest Precondition theory (§3.2). We then turn to the expression relation (§3.3), the fundamental theorem of logical relations, and the soundness theorem (§4).

3.1 Value Relations

The binary value relation is an Iris relation of type $Rel \triangleq Val \times Val \to iProp_{\square}$ where $iProp_{\square}$ denotes the class of *persistent propositions* (described below) in Iris:

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

Similarly, the unary value relation is an Iris predicate of type $Pred \triangleq Val \to iProp_{\square}$.

By default, since Iris is a separation logic, propositions denote sets of resources and $P * Q$ holds for resources that can be split into two disjoint parts satisfying P and Q, respectively. The proposition $P \mathrel{-\!\!*} Q$ describes those resources which, if we combine them with a disjoint resource satisfying P, satisfies Q. As such, Iris propositions assert *ownership* of ephemeral (non-persistent) resources. For example, the points-to connectives $\iota \mapsto_L v$ and $\iota \mapsto_R v$ asserts exclusive ownership of location

$$\textsc{T-Var}\ \frac{x : \tau \in \Gamma}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} x : \tau} \qquad \textsc{T-unit}\ \frac{}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} () : 1^{\bot}} \qquad \textsc{T-bool}\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} b : \mathbb{B}^{\bot}} \qquad \textsc{T-nat}\ \frac{n \in \mathbb{N}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} n : \mathbb{N}^{\bot}}$$

$$\textsc{T-binop}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \mathbb{N}^{\ell_1} \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \mathbb{N}^{\ell_2} \quad \circledcirc : \mathbb{N} \times \mathbb{N} \Rightarrow t}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 \circledcirc e_2 : t^{\ell_1 \sqcup \ell_2}} \qquad \textsc{T-lam}\ \frac{\Xi \mid \Psi \mid \Gamma, x : \tau_1 \vdash_{\ell_e} e : \tau_2}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \lambda x.\, e : \left(\tau_1 \xrightarrow{\ell_e} \tau_2\right)^{\bot}}$$

$$\textsc{T-app}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \left(\tau_1 \xrightarrow{\ell_e} \tau_2\right)^{\ell} \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau_1 \quad \Psi \vdash \tau_2 \searrow \ell \quad \Psi \vdash pc \sqcup \ell \sqsubseteq \ell_e}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1\ e_2 : \tau_2}$$

$$\textsc{T-tlam}\ \frac{\Xi, \alpha \mid \Psi \mid \Gamma \vdash_{\ell_e} e : \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \Lambda e : \left(\forall_{\ell_e}\, \alpha.\, \tau\right)^{\bot}} \qquad \textsc{T-llam}\ \frac{\Xi \mid \Psi, \kappa \mid \Gamma \vdash_{\ell_e} e : \tau \quad \mathrm{FV}(\ell_e) \subseteq \Psi \cup \{\kappa\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \Lambda e : \left(\forall_{\ell_e}\, \kappa.\, \tau\right)^{\bot}}$$

$$\textsc{T-tapp}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \left(\forall_{\ell_e}\, \alpha.\, \tau\right)^{\ell} \quad \Psi \vdash pc \sqcup \ell \sqsubseteq \ell_e \quad \mathrm{FV}(t) \subseteq \Xi}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e\ \_ : \tau[t/\alpha]}$$

$$\textsc{T-lapp}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \left(\forall_{\ell_e}\, \kappa.\, \tau\right)^{\ell} \quad \Psi \vdash pc \sqcup \ell \sqsubseteq \ell_e[\ell'/\kappa] \quad \Psi \vdash \tau[\ell'/\kappa] \searrow \ell \quad \mathrm{FV}(\ell') \subseteq \Psi}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e\ \_ : \tau[\ell'/\kappa]}$$

$$\textsc{T-if}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \mathbb{B}^{\ell} \quad \forall i \in \{1,2\}\,.\, \Xi \mid \Psi \mid \Gamma \vdash_{pc \sqcup \ell} e_i : \tau \quad \Psi \vdash \tau \searrow \ell}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \textsc{T-inj}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau_i \quad i \in \{1,2\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{inj}_i\ e : (\tau_1 + \tau_2)^{\bot}}$$

$$\textsc{T-pair}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \tau_1 \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau_2}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\bot}} \qquad \textsc{T-proj}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\ell} \quad \Psi \vdash \tau_i \searrow \ell \quad i \in \{1,2\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \pi_i\ e : \tau_i}$$

$$\textsc{T-match}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\ell} \quad \forall i \in \{1,2\}\,.\, \Xi \mid \Psi \mid \Gamma, x : \tau_i \vdash_{pc \sqcup \ell} e_i : \tau \quad \Psi \vdash \tau \searrow \ell}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inj}_i\ \Rightarrow e_i\ \mathsf{end} : \tau}$$

$$\textsc{T-fold}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau[\mu\, \alpha.\, \tau/\alpha]}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{fold}\ e : (\mu\, \alpha.\, \tau)^{\bot}} \qquad \textsc{T-unfold}\ \frac{\Psi \vdash \tau[\mu\, \alpha.\, \tau/\alpha] \searrow \ell \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\mu\, \alpha.\, \tau)^{\ell}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{unfold}\ e : \tau[\mu\, \alpha.\, \tau/\alpha]}$$

$$\textsc{T-pack}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau[t/\alpha]}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{pack}\ e : (\exists \alpha.\, \tau)^{\bot}}$$

$$\textsc{T-unpack}\ \frac{\Psi \vdash \tau \searrow \ell \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{pack}\ e_1 : (\exists \alpha.\, \tau')^{\ell} \quad \Xi, \alpha \mid \Psi \mid \Gamma, x : \tau' \vdash_{pc \sqcup \ell} e_2 : \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{unpack}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{T-alloc}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \quad \Psi \vdash \tau \searrow pc}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{ref}(e) : \mathsf{ref}(\tau)^{\bot}} \qquad \textsc{T-store}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \mathsf{ref}(\tau)^{\ell} \quad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau \quad \Psi \vdash \tau \searrow pc \sqcup \ell}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 \leftarrow e_2 : 1^{\bot}}$$

$$\textsc{T-load}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{ref}(e_1) : \mathsf{ref}(\tau)^{\ell} \quad \Xi \mid \Psi \vdash \tau <: \tau' \quad \Psi \vdash \tau' \searrow \ell}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} !\, e : \tau'}$$

$$\textsc{T-sub}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc'} e : \tau' \quad \Psi \vdash pc \sqsubseteq pc' \quad \Xi \mid \Psi \vdash \tau' <: \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau}$$

Fig. 1. Typing relation.

$$\textsc{S-refl}\ \frac{\mathrm{FV}(t) \subseteq \Xi}{\Xi \mid \Psi \vdash t <: t} \qquad \textsc{S-trans}\ \frac{\Xi \mid \Psi \vdash t_1 <: t_2 \qquad \Xi \mid \Psi \vdash t_2 <: t_3}{\Xi \mid \Psi \vdash t_1 <: t_3}$$

$$\textsc{S-arrow}\ \frac{\Xi \mid \Psi \vdash \tau_1' <: \tau_1 \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2' \qquad \Psi \vdash \ell_2 \sqsubseteq \ell_1}{\Xi \mid \Psi \vdash \tau_1 \xrightarrow{\ell_1} \tau_2 <: \tau_1' \xrightarrow{\ell_2} \tau_2'} \qquad \textsc{S-tforall}\ \frac{\Psi \vdash \ell_2 \sqsubseteq \ell_1 \qquad \Xi, \alpha \mid \Psi \vdash \tau_1 <: \tau_2}{\Xi \mid \Psi \vdash \forall_{\ell_1}\, \alpha.\, \tau_1 <: \forall_{\ell_2}\, \alpha.\, \tau_2}$$

$$\textsc{S-lforall}\ \frac{\Psi, \kappa \vdash \ell_2 \sqsubseteq \ell_1 \qquad \Xi \mid \Psi, \kappa \vdash \tau_1 <: \tau_2}{\Xi \mid \Psi \vdash \forall_{\ell_1}\, \kappa.\, \tau_1 <: \forall_{\ell_2}\, \kappa.\, \tau_2} \qquad \textsc{S-prod}\ \frac{\Xi \mid \Psi \vdash \tau_1 <: \tau_1' \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2'}{\Xi \mid \Psi \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

$$\textsc{S-sum}\ \frac{\Xi \mid \Psi \vdash \tau_1 <: \tau_1' \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2'}{\Xi \mid \Psi \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'} \qquad \textsc{S-labeled}\ \frac{\Psi \vdash \ell_1 \sqsubseteq \ell_2 \qquad \Xi \mid \Psi \vdash t_1 <: t_2}{\Xi \mid \Psi \vdash t_1{}^{\ell_1} <: t_2{}^{\ell_2}}$$

Fig. 2. Subtyping relation.

$$\begin{aligned}
\sigma ::= {} & 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{iProp} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \ldots && \text{(Types)}\\
P, Q ::= {} & x \mid \lambda x : \sigma.\, t \mid t(u) \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q && \text{(Propositional logic)}\\
\mid {} & \forall x : \sigma.\, P \mid \exists x : \sigma.\, P \mid t = u && \text{(Higher-order logic)}\\
\mid {} & P * Q \mid P \mathrel{-\!\!*} Q \mid \iota \mapsto_L v \mid \iota \mapsto_R v \mid \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}}\, e\, \{\Phi\} && \text{(Separation logic)}\\
\mid {} & \Box P \mid \triangleright P \mid \mu x : \sigma.\, t \mid {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \mid \boxed{P}^{\mathcal{N}} \mid \ldots && \text{(Iris-specific connectives)}
\end{aligned}$$

Fig. 3. Syntax of Iris. t and u represent arbitrary terms.

ι storing value v in the state of the programs on the left- and right-hand side, respectively. Such proposition may cease to hold, *e.g.*, when ι is updated to point to some other value than v. Intuitively, persistent propositions are propositions that do *not* assert exclusive ownership of resources and once they hold, they hold forever. In Iris, this is expressed using the *persistence modality* $\Box$. The proposition $\Box P$ (read "persistently P") says P holds without asserting any ephemeral propositions and thus P can be freely duplicated, *i.e.*, $\Box P \vdash \Box P * \Box P$, and eliminated, *i.e.*, $\Box P \vdash P$. It is important that our value relations are defined using persistent predicates as our type system is intuitionistic, in the sense that it admits the usual structural rules, which, *e.g.*, means that the assumption that a value has a type τ may be used repeatedly.

Binary value relation. The binary value relations $[\![\tau]\!]^{\rho}_{\Theta}$ and $[\![t]\!]^{\rho}_{\Theta}$ for a labeled type τ and an unlabeled type t are defined by mutual induction on τ and t. Here $\rho : \mathit{LabelVar} \to \mathcal{L}$ is a semantic label environment mapping label variables to labels, and Θ is a semantic type environment for type variables, as is usual for interpretations of languages with parametric polymorphism. However, for every type variable we keep both a binary relation and two unary relations, one for each of the two sides:

$$\Theta : \mathit{TypeVar} \to \mathit{Rel} \times \mathit{Pred} \times \mathit{Pred}.$$

We use $\Theta_L, \Theta_R : \mathit{TypeVar} \to \mathit{Pred}$ as shorthand for $\pi_2 \circ \Theta$ and $\pi_3 \circ \Theta$, respectively, where $\pi_i(x)$ denotes the ith projection of x. It will be a property of the binary relation that the following

binary-unary subsumption property (Lemma 4.2) holds:

$$\forall v, v'.\, [\![\tau]\!]^{\rho}_{\Theta}(v, v') \twoheadrightarrow [\![\tau]\!]^{\rho}_{\Theta_L}(v) * [\![\tau]\!]^{\rho}_{\Theta_R}(v'),$$

where $[\![\tau]\!]^{\rho}_{\Theta_L}(v)$ and $[\![\tau]\!]^{\rho}_{\Theta_R}(v')$ denote the unary interpretation of τ at v and v'. This property is crucial: Intuitively, even though two values are observationally *equivalent* to a ζ observer, they are also—independently—not supposed to have any latent ζ-observable side-effects. We elaborate further on this in §4 with more technical details. However, for the property to hold, the binary value relation has to be set up carefully, and the binary relation on open terms (explained in §3.3) requires that Θ is *coherent* in the following sense:

$$Coh(\Theta) \triangleq \mathop{\Large *}_{(\Phi,\Phi_L,\Phi_R)\in \mathrm{Im}(\Theta)} \square\, (\forall v, v'.\, \Phi(v, v') \twoheadrightarrow \Phi_L(v) * \Phi_R(v'))\,.$$

The big iterated separating conjunction quantifies over all triples (Φ, Φ_L, Φ_R) in the image of Θ and demands that the binary-unary subsumption property holds for the relations. The definition of the value relations, which we now explain, is shown in Figure 4.

The value interpretation of *labeled* types makes use of an interpretation $[\![\ell]\!]_\rho$ of syntactic labels ℓ defined as follows:

$$[\![\kappa]\!]_\rho \triangleq \rho(\kappa)$$
$$[\![l]\!]_\rho \triangleq l$$
$$[\![\ell_1 \sqcup \ell_2]\!]_\rho \triangleq [\![\ell_1]\!]_\rho \sqcup [\![\ell_2]\!]_\rho.$$

As above, ρ is an environment mapping label variables to labels. Notice that in the last equation, the $\sqcup$ on the left is the formal syntactic least upper bound whereas the $\sqcup$ on the right is the least upper bound in the lattice $\mathcal{L}$.

The interpretation of labeled types now follows the intuition given in the beginning of this section: low-labeled types (where $[\![\ell]\!]_\rho \sqsubseteq \zeta$) are distinguishable to the observer, and thus values should be related by the binary relation; high-labeled types (where $[\![\ell]\!]_\rho \not\sqsubseteq \zeta$) are *in*distinguishable to the observer and thus values should individually satisfy the unary interpretation to ensure that any latent effects will not be ζ-observable.

$$[\![t^{\ell}]\!]^{\rho}_{\Theta}(v, v') \triangleq \begin{cases} [\![t]\!]^{\rho}_{\Theta}(v, v') & \text{if } [\![\ell]\!]_\rho \sqsubseteq \zeta \\ [\![t]\!]^{\rho}_{\Theta_L}(v) * [\![t]\!]^{\rho}_{\Theta_R}(v') & \text{if } [\![\ell]\!]_\rho \not\sqsubseteq \zeta \end{cases}$$

This is the key point of interaction between the unary and binary relation.

The value interpretation of *unlabeled* types follows a structure that readers familiar with previous logical-relations models in Iris will find familiar. However, we also need to guarantee that the relation satisfies the binary-unary subsumption property.

If t is an (unlabeled) ground type (1, $\mathbb{B}$, or $\mathbb{N}$), two values are related at t if they are equal and compatible with the type. For products $\tau_1 \times \tau_2$, two values are related if they are both pairs with components related at their respective types. Similarly for sums $\tau_1 + \tau_2$, two values are related if they are both inj_i for the same i and their contents are related at τ_i.

The first clause of the interpretation of the function type is a slight variation of the classical function type interpretation in logical-relations models: two values v and v' are related at type $\tau_1 \xrightarrow{\ell_e} \tau_2$ if they map inputs related at τ_1 to related results in the expressions interpretation of τ_2. Note that we wrap this clause in a persistence modality in order to ensure that the relation is persistent and that we ignore the latent effect label. The latter will only be important for the unary interpretation. The two following clauses require that v and v' individually satisfy the unary interpretation; this is to ensure that the binary-unary subsumption property holds.

Value relation

$$\llbracket \alpha \rrbracket^{\rho}_{\Theta} \triangleq \pi_1\,(\Theta(\alpha))$$

$$\llbracket 1 \rrbracket^{\rho}_{\Theta}(v, v') \triangleq v = v' = ()$$

$$\llbracket \mathbb{B} \rrbracket^{\rho}_{\Theta}(v, v') \triangleq v = v' \in \{\mathsf{true}, \mathsf{false}\}$$

$$\llbracket \mathbb{N} \rrbracket^{\rho}_{\Theta}(v, v') \triangleq v = v' \in \mathbb{N}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \exists v_1, v_2, v_1', v_2'.\, v = (v_1, v_2) * v' = (v_1', v_2') * \llbracket \tau_1 \rrbracket^{\rho}_{\Theta}(v_1, v_1') * \llbracket \tau_2 \rrbracket^{\rho}_{\Theta}(v_2, v_2')$$

$$\llbracket \tau_1 + \tau_2 \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \bigvee_{i \in \{1,2\}} \exists w, w'.\, v = \mathsf{inj}_i\, w * v' = \mathsf{inj}_i\, w' * \llbracket \tau_i \rrbracket^{\rho}_{\Theta}(w, w')$$

$$\llbracket \tau_1 \xrightarrow{\ell_e} \tau_2 \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \Box\left(\forall w, w'.\, \llbracket \tau_1 \rrbracket^{\rho}_{\Theta}(w, w') \mathrel{-\!\!*} \mathcal{E}\llbracket \tau_2 \rrbracket^{\rho}_{\Theta}(v\ w, v'\ w')\right) * \llbracket \tau_1 \xrightarrow{\ell_e} \tau_2 \rrbracket^{\rho}_{\Theta_L}(v) * \llbracket \tau_1 \xrightarrow{\ell_e} \tau_2 \rrbracket^{\rho}_{\Theta_R}(v')$$

$$\llbracket \forall_{\ell_e}\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \Box\,(\forall \Phi : \mathit{Rel}.\, \forall \Phi_L, \Phi_R : \mathit{Pred}.\, \Box\left(\forall v, v'.\, \Phi(v, v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')\right) \mathrel{-\!\!*} \mathcal{E}\llbracket \tau \rrbracket^{\rho}_{\Theta, \alpha \mapsto (\Phi, \Phi_L, \Phi_R)}(v\ \_, v'\ \_)) * \llbracket \forall_{\ell_e}\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta_L}(v) * \llbracket \forall_{\ell_e}\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta_R}(v')$$

$$\llbracket \forall_{\ell_e}\, \kappa.\, \tau \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \Box\left(\forall l \in \mathcal{L}.\, \mathcal{E}\llbracket \tau \rrbracket^{\rho, \kappa \mapsto l}_{\Theta}(v\ \_, v'\ \_)\right) * \llbracket \forall_{\ell_e}\, \kappa.\, \tau \rrbracket^{\rho}_{\Theta_L}(v) * \llbracket \forall_{\ell_e}\, \kappa.\, \tau \rrbracket^{\rho}_{\Theta_R}(v')$$

$$\llbracket \exists \alpha.\, \tau \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \Box\,(\exists \Phi : \mathit{Rel}.\, \exists \Phi_L, \Phi_R : \mathit{Pred}.\, \Box\left(\forall v, v'.\, \Phi(v, v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')\right) * \exists w, w'.\, v = \mathsf{pack}\, w * v' = \mathsf{pack}\, w' * \llbracket \tau \rrbracket^{\rho}_{\Theta, \alpha \mapsto (\Phi, \Phi_L, \Phi_R)}(w, w'))$$

$$\llbracket \mu\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta} \triangleq \mu\, \Phi : \mathit{Rel}.\, \lambda(v, v').\exists w, w'.\, v = \mathsf{fold}\, w * v' = \mathsf{fold}\, w' * \triangleright \llbracket \tau \rrbracket^{\rho}_{\Theta, \alpha \mapsto (\Phi, \llbracket \mu\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta_L}, \llbracket \mu\, \alpha.\, \tau \rrbracket^{\rho}_{\Theta_R})}(w, w')$$

$$\llbracket \mathsf{ref}(\tau) \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \exists \iota, \iota'.\, v = \iota * v' = \iota' * \boxed{\exists w, w'.\, \iota \mapsto_L w * \iota' \mapsto_R w' * \llbracket \tau \rrbracket^{\rho}_{\Theta}(w, w')}^{\mathcal{N}_{root}.(\iota, \iota')}$$

$$\llbracket t^{\ell} \rrbracket^{\rho}_{\Theta}(v, v') \triangleq \begin{cases} \llbracket t \rrbracket^{\rho}_{\Theta}(v, v') & \text{if } \llbracket \ell \rrbracket_{\rho} \sqsubseteq \zeta \\ \llbracket t \rrbracket^{\rho}_{\Theta_L}(v) * \llbracket t \rrbracket^{\rho}_{\Theta_R}(v') & \text{if } \llbracket \ell \rrbracket_{\rho} \not\sqsubseteq \zeta \end{cases}$$

Expression relation

$$\mathcal{E}\llbracket \tau \rrbracket^{\rho}_{\Theta}(e, e') \triangleq \mathsf{mwp}\ e \sim e' \{\llbracket \tau \rrbracket^{\rho}_{\Theta}\}$$

Environment relation

$$\mathcal{G}\llbracket \cdot \rrbracket^{\rho}_{\Theta}(\epsilon, \epsilon) \triangleq \mathsf{True}$$

$$\mathcal{G}\llbracket \Gamma, x : \tau \rrbracket^{\rho}_{\Theta}(\vec{v}\ w, \overrightarrow{v'}\ w') \triangleq \mathcal{G}\llbracket \Gamma \rrbracket^{\rho}_{\Theta}(\vec{v}, \overrightarrow{v'}) * \llbracket \tau \rrbracket^{\rho}_{\Theta}(w, w')$$

Semantic typing judgment

$$\mathit{Coh}(\Theta) \triangleq \mathop{\Large *}_{(\Phi, \Phi_L, \Phi_R) \in \Theta} \Box\left(\forall v, v'.\, \Phi(v, v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')\right)$$

$$\Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e' : \tau \triangleq \Box\left(\begin{array}{l}\forall \Theta, \rho, \vec{v}, \overrightarrow{v'}.\, \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Theta) * \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(\rho) \mathrel{-\!\!*} \\ \qquad \mathit{Coh}(\Theta) * \mathcal{G}\llbracket \Gamma \rrbracket^{\rho}_{\Theta}(\vec{v}, \overrightarrow{v'}) \mathrel{-\!\!*} \mathcal{E}\llbracket \tau \rrbracket^{\rho}_{\Theta}(e[\vec{v}/\vec{x}], e'[\overrightarrow{v'}/\vec{x}])\end{array}\right)$$

Fig. 4. Binary interpretations.

For type-polymorphic types we use the semantic type environment which maps type variables to triples consisting of an Iris relation on values and two unary relations. We define the interpretation $[\![\alpha]\!]^{\rho}_{\Theta}$ of type variable α by looking up the variable in Θ and taking the first projection.

Universal types are interpreted using logical propositions that are also universally quantified but over *semantic predicates*, heavily relying on Iris's support for higher-order impredicative quantification. However, we quantify not only over a binary relation but also two unary relations for which we require the subsumption property to hold. This ensures that the semantic type environment is coherent. Two value v and v' are then related at type $\forall_{\ell_e}\, \alpha.\, \tau$ when type applications $v\ \_$ and $v'\ \_$ are related at τ in a semantic environment mapping α to the binary relation and the two unary relations. As in the case for the function type, we also require that v and v' satisfy the unary interpretation.

Label abstraction is interpreted following a similar pattern: We quantify over *semantic labels*—which are just labels from the lattice $\mathcal{L}$—and express that v and v' are related at type $\forall\!\!/_{\ell_e}\, \kappa.\, \tau$ when the applications $v\ \_$ and $v'\ \_$ are related at type τ in an extended semantic label environment mapping κ to the label l.

The interpretation of existential types $\exists \alpha.\, \tau$ quantifies existentially over a binary relation and two unary relations satisfying the subsumption property and relates values of the form pack w and pack w' if w and w' are related at type τ in an extended semantic type environment.

To interpret recursive types we make use of Iris's *guarded recursive predicates*. The guarded fixed-point operator $\mu\, x : \sigma.\, t$ of Iris can be used to define recursive predicates (without restrictions on variance for occurrences of x) by requiring that all recursive occurrences of x are *guarded* by a *later modality* $\triangleright$. Intuitively, the later modality asserts that something holds "one step of computation later". It is monotone ($P \vdash Q$ implies $\triangleright P \vdash \triangleright Q$) and can be introduced ($P \vdash \triangleright P$). In Iris, with the restriction of guardedness, the fixed-point operator satisfies the expected equation: $\mu\, x : \sigma.\, t = t[\mu\, x : \sigma.\, t/x]$. The key proof principle associated with the later modality is the Löb rule: $(\triangleright P \Rightarrow P) \Rightarrow P$, which is used to prove the binary-unary subsumption property (Lemma 4.2) in the case of recursive types.

Using this fixed-point property, two values are related at type $\mu\, \alpha.\, \tau$ if they are of the form fold w and fold w', and if w and w' are related at τ (under a later modality) in an extended type environment mapping α to the triple $([\![\mu\, \alpha.\, \tau]\!]^{\rho}_{\Theta}, [\![\mu\, \alpha.\, \tau]\!]^{\rho}_{\Theta_L}, [\![\mu\, \alpha.\, \tau]\!]^{\rho}_{\Theta_R})$. The unary relations again ensure that the extended semantic type environment is coherent.

Recall that the binary relation is intended to relate terms that are observationally equivalent to a ζ-observer. Hence related values of reference type $\mathsf{ref}(\tau)$ should be locations ι and ι' such that their contents may change but the contents should always stay related at type τ. To express this requirement, we make use of Iris's *invariant assertion* $\boxed{P}^{\mathcal{N}}$, which expresses the (persistent) knowledge that a proposition P holds at all times. In order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion, Iris features *invariant namespaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$. Iris annotates each invariant $\boxed{P}^{\mathcal{N}}$ with a namespace $\mathcal{N}$ to identify the invariant and, as we shall explain later, we annotate modal weakest preconditions $\mathsf{mwp}^{\mathcal{M}}_{\mathcal{E}}\, e\, \{\Phi\}$ with a mask $\mathcal{E}$ to keep track of which invariants are enabled and may be opened. If the mask is omitted we consider the modal weakest precondition with mask $\top$, the set of all invariant names.

In order to work with invariants formally in Iris we make use of the *update modality* ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}$. We write $\Rrightarrow_{\mathcal{E}}$ if $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$. Akin to how a weakest precondition is used to reason about physical state, the update modality is used to reason about ghost state. The update modality is annotated with masks $\mathcal{E}_1$ and $\mathcal{E}_2$ that denote which invariants are enabled and may be opened before and after the modality is introduced. Intuitively, the proposition ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ holds for resources that (given

the invariants in $\mathcal{E}_1$ are enabled) can be updated to resources that satisfy P (with the invariants in $\mathcal{E}_2$ enabled) without violating the environment's knowledge or ownership of resources. We write $P \;{}^{\mathcal{E}_1}\!\!\equiv\!\!*^{\mathcal{E}_2}\; Q$ as a shorthand for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q$.

Some formal rules for invariants in Iris can be found in Figure 5. An invariant $\boxed{P}^{\mathcal{N}}$ can be

$$\textsc{inv-alloc}\quad \triangleright P \vdash \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} \qquad \textsc{inv-persist}\quad \boxed{P}^{\mathcal{N}} \vdash \Box \boxed{P}^{\mathcal{N}} \qquad \textsc{inv-open-upd}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\triangleright P * Q)) \vdash \Rrightarrow_{\mathcal{E}} Q}$$

Fig. 5. Rules for invariants.

allocated (INV-ALLOC) by giving up ownership of P, a possibly ephemeral proposition. Invariants are persistent (INV-PERSIST) and hence duplicable. The contents of invariants may be accessed in a carefully restricted way (INV-OPEN-UPD): to prove $\Rrightarrow_{\mathcal{E}} Q$, we may open an invariant and assume $\triangleright P$ as long as we re-establish the invariant $\triangleright P$. For more details on invariants in Iris, including the role of the later modality in the rules, see Birkedal and Bizjak [2017]; Jung et al. [2018b].

With the invariant connective at hand, the binary relation for reference types $\mathsf{ref}(\tau)$ is straightforward and relates locations that invariantly have contents related at type τ. Here $\mathcal{N}_{root}.(\iota, \iota')$ is the namespace designated to the invariant on the locations ι and ι'.

Unary value relation. The unary value relations $[\![\tau]\!]^{\rho}_{\Delta}$ and $[\![t]\!]^{\rho}_{\Delta}$ for a labeled type τ and an unlabeled type t are defined by mutual induction on τ and t; however, the label on labeled types is ignored since, as mentioned earlier, the point of the unary relation is to ensure that computations embedded in values have no ζ-observable side-effects.

$$[\![t^{\ell}]\!]^{\rho}_{\Delta}(v) \triangleq [\![t]\!]^{\rho}_{\Delta}(v).$$

Here Δ is a semantic type environment mapping type variables to unary relations of type *Pred* and ρ is a semantic label environment mapping label variables to labels. The full relation, which we now explain, is shown in Figure 6.

The only values of ground type are values compatible with the type. Similarly, values of type $\tau_1 \times \tau_2$ are pairs with components inhabiting the interpretation of τ_1 and τ_2, respectively. Values of type $\tau_1 + \tau_2$ are inj_i with contents related at τ_i.

The unary interpretation of function type $\tau_1 \xrightarrow{\iota_e} \tau_2$ follows the canonical pattern and takes related input at τ_1 to related results at τ_2. However, notice that the unary expression relation is indexed with the latent effect label of the function. The unary relation is only concerned with expressions in high-labeled contexts; low-labeled contexts are ζ-observable and the unary relation poses no requirements on these. We will return to these matters in §3.3 when discussing the expression relations.

Both type- and label-polymorphic types are interpreted by quantifying over their semantic counterparts and a value v inhabits the polymorphic type if application v _ inhabits τ in the extended semantic environment. As for function types, the expression relation is indexed with the latent effect label of the polymorphic binder. The interpretation of existential types and recursive types follows the same pattern as in the binary interpretation.

The interpretation of reference types is the central part of the unary interpretation and states that terms have no ζ-observable side-effects. Intuitively, a reference containing data with a label lower than ζ is *not* allowed to change when execution conditionally depends on higher-labeled information as this would implicitly leak the high-labeled information through the state. The

Value relation

$$
\begin{aligned}
[\![\alpha]\!]^{\rho}_{\Delta} &\triangleq \Delta(\alpha)\\
[\![1]\!]^{\rho}_{\Delta}(v) &\triangleq v = ()\\
[\![\mathbb{B}]\!]^{\rho}_{\Delta}(v) &\triangleq v \in \{\mathsf{true}, \mathsf{false}\}\\
[\![\mathbb{N}]\!]^{\rho}_{\Delta}(v) &\triangleq v \in \mathbb{N}\\
[\![\tau_1 \times \tau_2]\!]^{\rho}_{\Delta}(v) &\triangleq \exists v_1, v_2.\, v = (v_1, v_2) * [\![\tau_1]\!]^{\rho}_{\Delta}(v_1) * [\![\tau_2]\!]^{\rho}_{\Delta}(v_2)\\
[\![\tau_1 + \tau_2]\!]^{\rho}_{\Delta}(v) &\triangleq \bigvee_{i\in\{1,2\}} \exists w.\, v = \mathsf{inj}_i\, w * [\![\tau_i]\!]^{\rho}_{\Delta}(w)\\
[\![\tau_1 \xrightarrow{\ell_e} \tau_2]\!]^{\rho}_{\Delta}(v) &\triangleq \Box\left(\forall w.\, [\![\tau_1]\!]^{\rho}_{\Delta}(w) \mathrel{-\!\!*} \mathcal{E}_{\ell_e}[\![\tau_2]\!]^{\rho}_{\Delta}(v\ w)\right)\\
[\![\forall_{\ell_e}\, \alpha.\, \tau]\!]^{\rho}_{\Delta}(v) &\triangleq \Box\left(\forall f : \mathit{Pred}.\, \mathcal{E}_{\ell_e}[\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto f}(v\ \_)\right)\\
[\![\forall_{\ell_e}\, \kappa.\, \tau]\!]^{\rho}_{\Delta}(v) &\triangleq \Box\left(\forall l \in \mathcal{L}.\, \mathcal{E}_{\ell_e}[\![\tau]\!]^{\rho,\kappa\mapsto l}_{\Delta}(v\ \_)\right)\\
[\![\exists\alpha.\, \tau]\!]^{\rho}_{\Delta}(v) &\triangleq \Box\left(\exists \Phi : \mathit{Pred}.\, \exists w.\, v = \mathsf{pack}\, w * [\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto\Phi}(w)\right)\\
[\![\mu\,\alpha.\, \tau]\!]^{\rho}_{\Delta} &\triangleq \mu\,\Phi : \mathit{Pred}.\, \lambda v. \exists w.\, v = \mathsf{fold}\, w * \triangleright [\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto f}(w)\\
[\![\mathsf{ref}(t^{\ell})]\!]^{\rho}_{\Delta}(v) &\triangleq \exists \iota, \mathcal{N}.\, v = \iota * \mathcal{R}(\Delta, \rho, \iota, \ell, \mathcal{N})\\
\mathcal{R}(\Delta, \rho, \iota, \ell, \mathcal{N}) &\triangleq \begin{cases}
\Box\forall\mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rrightarrow \\
\quad\left(\mathcal{E} \Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright\left(\begin{array}{l}\exists w.\, \iota \mapsto_i w * [\![\tau]\!]^{\rho}_{\Delta}(w) *\\ \left(\left(\triangleright\, \iota \mapsto_i w * [\![\tau]\!]^{\rho}_{\Delta}(w)\right)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}}\ \mathsf{True}\right)\end{array}\right)\right) & \text{if } [\![\ell]\!]_{\rho} \sqsubseteq \zeta\\
\Box\forall\mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rrightarrow \\
\quad\left(\mathcal{E} \Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright\left(\begin{array}{l}\exists w'.\, \iota \mapsto_i w' * [\![\tau]\!]^{\rho}_{\Delta}(w') *\\ \left(\left(\triangleright\, \exists w'.\, \iota \mapsto_i w' * [\![\tau]\!]^{\rho}_{\Delta}(w')\right)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}}\ \mathsf{True}\right)\end{array}\right)\right) & \text{if } [\![\ell]\!]_{\rho} \not\sqsubseteq \zeta
\end{cases}\\
[\![t^{\ell}]\!]^{\rho}_{\Delta}(v) &\triangleq [\![t]\!]^{\rho}_{\Delta}(v)
\end{aligned}
$$

Expression relation

$$
\mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e) \triangleq [\![pc]\!]_{\rho} \not\sqsubseteq \zeta \Rightarrow \mathsf{mwp}^{\mathcal{M}}{}_{\Rrightarrow}\ e\ \{[\![\tau]\!]^{\rho}_{\Delta}\}
$$

Environment relation

$$
\begin{aligned}
\mathcal{G}[\![\cdot]\!]^{\rho}_{\Delta}(\epsilon) &\triangleq \mathsf{True}\\
\mathcal{G}[\![\Gamma, x : \tau]\!]^{\rho}_{\Delta}(\vec{v}\ w) &\triangleq \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Delta}(\vec{v}) * [\![\tau]\!]^{\rho}_{\Delta}(w)
\end{aligned}
$$

Semantic typing judgment

$$
\Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau \triangleq \Box\left(\begin{array}{c}\forall \Delta, \rho, \vec{v}\,.\, \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Delta) * \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(\rho) \mathrel{-\!\!*}\\ \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Delta}(\vec{v}) \mathrel{-\!\!*} \mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e[\vec{v}/\vec{x}])\end{array}\right)
$$

Fig. 6. Unary interpretations.

contents of references with a label higher than ζ, however, can always be modified as long as the new contents are compatible with the types.

In order to state this intuition formally in Iris, *while* at the same time ensuring that the binary-unary subsumption property holds, we make use of the update modality to encode a relaxed form of semantic invariants. Instead of using an Iris invariant to capture the meaning of a reference

type, we essentially use the key properties of Iris invariants (that they can be opened and closed again) and, depending on the label of the contents of the reference, we can express whether the value stored in the reference is allowed to change or not. As such, values v of type $\mathsf{ref}(t^\ell)$ are locations for which there exists a namespace $\mathcal{N}$ such that $\mathcal{R}(\Delta, \rho, \iota, \ell, \mathcal{N})$ holds. The namespace $\mathcal{N}$ is *some* namespace associated with ι. The $\mathcal{R}(\Delta, \rho, \iota, \ell, \mathcal{N})$ proposition states that if the content of the reference is of a low-labeled type ($[\![\ell]\!]_\rho \sqsubseteq \zeta$) then the content of ι is not allowed to change in an observable way:

$$\square \forall \mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rrightarrow \left({}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l} \exists w.\, \iota \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) * \\ \left(\left(\triangleright\, \iota \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) \right) \;\; {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!*^{\mathcal{E}} \; \mathsf{True} \right) \end{array} \right) \right).$$

If we ignore the later modalities, intuitively, this says that if the namespace $\mathcal{N}$ is currently enabled we can, by disabling $\mathcal{N}$, get ownership of the points-to connective $\iota \mapsto_i w$ with $i \in \{L, R\}$ such that w inhabits $[\![\tau]\!]^\rho_\Delta$. Moreover, the namespace $\mathcal{N}$ can only be enabled again by giving back the ownership of the points-to connective with *unmodified* contents w.

In a similar fashion, if the content of the reference is of a high-labeled type ($[\![\ell]\!]_\rho \not\sqsubseteq \zeta$) then the content *is* allowed to change:

$$\square \forall \mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rrightarrow \left({}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l} \exists w.\, \iota \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) * \\ \left(\left(\triangleright\, \exists w'.\, \iota \mapsto_i w' * [\![\tau]\!]^\rho_\Delta(w') \right) \;\; {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!*^{\mathcal{E}} \; \mathsf{True} \right) \end{array} \right) \right).$$

Intuitively, as before, if the namespace $\mathcal{N}$ is currently enabled we can, by disabling $\mathcal{N}$, get ownership of the points-to connective $\iota \mapsto_i w$ such that w inhabits τ. However, we can enable $\mathcal{N}$ again by giving back the ownership of the points-to connective with *any* content w' as long as it still inhabits type τ.

3.2 Modal Weakest Precondition

We now turn to the theory of the *Modal Weakest Precondition (MWP)* connective. Recall from the Introduction that due its termination-insensitive nature, existing approaches using Iris' weakest preconditions do not suffices for defining our expression relations. Moreover, we will need both a binary and a unary connective that interact in a reasonable way. As will be clear by the end of this section, the MWP theory accommodates all of this while providing high-level reasoning principles.

Similarly to how the standard weakest precondition in Iris is defined [Krebbers et al. 2017a], our new definition of a modal weakest precondition is language agnostic; it is not tied to a particular programming language and it is defined generically over any suitable notion of expression, state, and reduction relation. As a consequence of this generality, we do not make any assumptions on how the state of the programming language is defined; instead, as for standard Iris weakest preconditions, we parameterize modal weakest preconditions by a *state interpretation* $S : \mathit{State} \to \mathit{iProp}$. The S predicate interprets the state of the programming language using Iris propositions, *e.g.*, as a resource for modeling the heap of the program.

The modal weakest precondition connective is also parameterized by a modal operator and, indeed, one of the key strengths of the the connective is its generality and the fact that instantiations of it automatically inherit a set of basic structural proof rules (*cf.* Figure 7) that hold irrespective of the particular modality and programming language. For a particular instantiation of the connective, one can then prove soundness of more specific proof rules, *e.g.*, for heap-manipulating operations (*cf.* Lemma 3.3) and for the interaction with other instantiations with different modal operators (*cf.* Lemma 3.7).

In this paper, we will use the generality of modal weakest preconditions to reason about the λ_{sec} programming language: we will use three different instantiations with three different modalities for our logical-relations model; in fact, one of these modalities will be defined in terms of an earlier instantiation. We start by giving a simplified presentation in §3.2.1 before giving the definition in its full generality in §3.2.2. Finally, we use the theory of modal weakest preconditions in the subsequent §3.3 to define and reason about the expression relations of our logical-relations model.

3.2.1 Modal Weakest Precondition (Simplified). We define a predicate $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}$ which intuitively says that if program e reduces to a value v in n steps then $\Phi(v, n)$ holds under modality M. The predicate is parameterized over a mask $\mathcal{E} \in \mathit{Masks} = \wp(\mathit{InvName})$ and the modality $\mathsf{M} : \mathit{Masks} \to \mathbb{N} \to \mathit{iProp} \to \mathit{iProp}$. The modality M is indexed by a mask $\mathcal{E}$ and a natural number n. The invariant names in $\mathcal{E}$ are those invariants the modality may allow to be opened, if the modality allows the use of invariants at all. The number n is the number of logical steps that the modality allows which we tie to the physical steps of the program execution in the definition of $\mathsf{mwp}^{\mathcal{M}}_{\mathcal{E}}\, e\, \{\Phi\}$; the preliminary definition is as follows:

$$\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq \forall \sigma_1, \sigma_2, v, n.\, (e, \sigma_1) \to^n (v, \sigma_2) \twoheadrightarrow S(\sigma_1) \twoheadrightarrow \mathsf{M}_{\mathcal{E};n}(\Phi(v, n) * S(\sigma_2)).$$

The predicate expresses that if (e, σ_1) reduces to (v, σ_2) in n steps and $S(\sigma_1)$ holds then under modality M both $\Phi(v, n)$ and $S(\sigma_2)$ will hold. Note that the predicate does not require that the program is *safe* to execute, nor that it terminates. In particular, if the program gets stuck or diverges then $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}$ holds trivially.

The connective can be used for a range of different modalities; we only require that the modality M satisfies two conditions:

$$\mathcal{E} \subseteq \mathcal{E}' \Rrightarrow P \twoheadrightarrow Q \vdash \mathsf{M}_{\mathcal{E};n}(P) \twoheadrightarrow \mathsf{M}_{\mathcal{E}';n}(Q) \qquad \text{(monotone)}$$

$$\mathsf{M}_{\mathcal{E};0}(P) \vdash \mathsf{M}_{\mathcal{E};n}(P) \qquad \text{(introducable)}$$

We say that the modality M is *valid* if it satisfies the two conditions.

Given a valid modality, the $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}$ predicate satisfies several general structural rules. We present a selection of such rules in Figure 7. Most of these are self-explanatory, but we point out the importance of the MWP-BIND rule which is crucial for local reasoning.

$$\frac{\forall \sigma.\, (\sigma, e) \to (\sigma', e') \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e'\, \{\Phi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}}\ \textsc{MWP-pure-step} \qquad \frac{\mathsf{M}_{\mathcal{E};0}(\Phi(v, 0))}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, v\, \{\Phi\}}\ \textsc{MWP-value}$$

$$\frac{\forall v, n.\, \Phi(v, n) \twoheadrightarrow \Psi(v, n) \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Psi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}}\ \textsc{MWP-mono} \qquad \frac{\mathcal{E} \subseteq \mathcal{E}' \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}'}\, e\, \{\Phi\}}\ \textsc{MWP-mask-mono}$$

$$\frac{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{v, n.\, \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, K[v]\, \{w, m.\, \Phi(w, n + m)\}\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, K[e]\, \{\Phi\}}\ \textsc{MWP-bind}$$

Fig. 7. Excerpt of rules for the modal weakest precondition connective given a valid modality. More rules, *e.g.*, for reasoning about single steps with state manipulation, can be found in the appendix.

Example 3.1 (MWP instance: Update modality). Let $\mathsf{M}^{\Rightarrow}_{\mathcal{E},n}(P) \triangleq \Rightarrow_{\mathcal{E}} P$. This is a valid modality. The modality does not allow any logical steps (and ignores its index n). When proving $\mathsf{mwp}^{\mathsf{M}^{\Rightarrow}}_{\mathcal{E}}\, e\, \{\Phi\}$,

however, all invariants in $\mathcal{E}$ may be opened before establishing the post condition Φ but must be immediately closed.

The simplified presentation given so far suffices for defining the modal weakest precondition instance that we will use for the unary expression interpretation. This is the point of the following example.

Example 3.2 (MWP instance: Step-taking update modality). Let

$$\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}(P) \triangleq \left({}^{\mathcal{E}}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\mathcal{E}}\right)^n \Rightarrow_{\mathcal{E}} P.$$

where $\left({}^{\mathcal{E}}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\mathcal{E}}\right)^n$ is n times repetition of ${}^{\mathcal{E}}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\mathcal{E}}$. The modality $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}$ is valid and can be thought of as a *step-taking update modality*. Intuitively, $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}(P)$ expresses that n steps into the future, we can update our resources to satisfy P, and, moreover, for every step, all invariants in $\mathcal{E}$ may be opened to reason about progress as long as they are immediately closed afterwards. In practice, the later modality allows stripping later modalities from assumptions that we get when opening invariants.

Using the structural rules for MWP in Figure 7, in particular the MWP-BIND rule, one can see that the modality distributes over compound expressions such that when proving $\mathsf{mwp}^{\mathsf{M}^{\Rightarrow\triangleright}}_{\mathcal{E}}\ e\ \{\Phi\}$, one is allowed to open invariants *atomically*, *i.e.*, for the duration of a single atomic step.

When instantiating the modal weakest precondition theory with λ_{sec} and the $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}$ modality we can derive the following properties for reasoning about heap-manipulating operations. Note that the proofs of these properties make use of the rules omitted from Figure 7.

LEMMA 3.3 (PROPERTIES OF STEP-TAKING UPDATE MWP WITH λ_{sec}).

(1) $\triangleright \forall \iota.\, \iota \mapsto_i v \twoheadrightarrow Q\ \iota \vdash \mathsf{mwp}^{\mathcal{M}_{\Rightarrow\triangleright}}_{\mathcal{E}}\ \mathsf{ref}(v)\ \{v.\ Q\}$

(2) $\triangleright \iota \mapsto_i v * \triangleright(\iota \mapsto_i v \twoheadrightarrow Q\ v) \vdash \mathsf{mwp}^{\mathcal{M}_{\Rightarrow\triangleright}}_{\mathcal{E}}\ !\iota\ \{v.\ Q\}$

(3) $\triangleright \iota \mapsto_i v * \triangleright(\iota \mapsto_i w \twoheadrightarrow Q\ ()) \vdash \mathsf{mwp}^{\mathcal{M}_{\Rightarrow\triangleright}}_{\mathcal{E}}\ \iota \leftarrow w\ \{v.\ Q\}$

Lemma 3.3 state properties that allow us to allocate, read, and modify the heap. They all express that the postcondition Q will hold if the resources needed are given and Q holds for the updated resources.

3.2.2 Modal Weakest Precondition (Full Definition). The definition of modal weakest precondition connective presented so far suffices for unary reasoning about programs. A specific instance of it has already been used in previous work by Timany et al. [2018] who considered an instantiation with a so-called future modality. However, in order to facilitate termination-insensitive reasoning about two programs at the same time, we generalize the definition further such that we can use an MWP connective *as the modality* of another MWP connective. In §3.3, we will see how this general connective is particularly useful for defining and working with our binary logical-relations model.

The key idea behind the generalization is to let the modality—apart from the number of steps of the execution and the mask—have its own "state" embodied in an index and to let the proposition that the modality acts on be parameterized over some information provided by the modality. For unary reasoning, both of these indices will just be the unit type, meaning the modality has no state and provides no information to the postcondition (in which case we recover the simplified presentation from the above). However, when used for binary reasoning, as in our binary logical-relations model, the index of the modality will be the second program, and the postcondition parameter will be the return value of the second program.

Formally, we parameterize the modality by two types, A and B, and a predicate BindCond that determines when and how the modality will change "when the binding lemma applies" (explained below). We bundle these parameters together with the modality $\mathsf{M} : A \to \mathit{Masks} \to \mathbb{N} \to (B \to \mathit{iProp}) \to \mathit{iProp}$ as a tuple $\mathcal{M}$.

Definition 3.4 (Modal Weakest Precondition). Let $\mathcal{M} = (A, B, \mathsf{M}, \mathsf{BindCond})$ and $a \in A$. Then

$$\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a}\, e\, \{\Phi\} \triangleq \forall \sigma_1, \sigma_2, v, n.\, (e, \sigma_1) \to^n (v, \sigma_2) \mathrel{-\!\!*} S(\sigma_1) \mathrel{-\!\!*} \mathsf{M}^a_{\mathcal{E};n}(\lambda b.\, \Phi(v, n, b) * S(\sigma_2)).$$

For the modality defined by $\mathcal{M}$ to be valid it has to satisfy the two conditions from above (monotonicity and introducability) and, moreover, whenever $\mathsf{BindCond}(a, a', f, g)$ holds, then we should also have

$$\mathsf{M}^{a'}_{\mathcal{E};n}(\lambda b.\, \mathsf{M}^{f(b)}_{\mathcal{E};m}(\lambda b'.\, \Phi(g(b, b')))) \vdash \mathsf{M}^a_{\mathcal{E};n+m}(\Phi). \qquad \text{(binding)}$$

Intuitively, $\mathsf{BindCond}(a, a', f, g)$ defines when and how the modality can be chained together through binding; a modality with index a and $n + m$ logical steps can be split into the sequence of the modality with index a' and n steps followed by the modality with index $f(b)$ and m steps given the postcondition is updated according to g. This will allow us to suitably generalize MWP-BIND to take into account the new indices and how the modality may evolve.

MWP-BIND-GEN

$$\frac{\mathsf{BindCond}(a, a', f, g) \qquad \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a'}\, e \left\{v, n, b.\, \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};f(b)}\, K[v]\, \{w, m, b'.\, \Phi(w, n + m, g(b, b'))\}\right\}}{\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a}\, K[e]\, \{\Phi\}}$$

The generalized modal connective allows us to use a modal weakest precondition connective as the modality of another modal weakest precondition. This not only allows us to define a relational predicate on two computations (as we will see below), but also to have a collection of proof rules (*cf.* Figure 7) for reasoning about the individual computations.

Example 3.5 (MWP instance: Binary step-taking update modality). The relational predicate used in our binary logical-relations model has the following shape when unfolding the definition:

$$\begin{aligned}\mathsf{mwp}_{\mathcal{E}}\, e_1 \sim e_2\, \{v, w.\, \Phi\} = \forall \sigma_1, \sigma_1', v, n.\, (e_1, \sigma_1) \to^n (v, \sigma_1') \mathrel{-\!\!*} S_1(\sigma_1) \mathrel{-\!\!*} \\ \forall \sigma_2, \sigma_2', w, m.\, (e_2, \sigma_2) \to^m (w, \sigma_2') \mathrel{-\!\!*} S_2(\sigma_2) \mathrel{-\!\!*} \\ \left({}^{\mathcal{E}}\Rrightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rrightarrow^{\mathcal{E}}\right)^{n+m} \Rrightarrow_{\mathcal{E}} \left(\Phi(v, w) * S_1(\sigma_1') * S_2(\sigma_2')\right) \end{aligned} \tag{4}$$

and is, as it seems, a binary version of the instance from Example 3.2. Intuitively, if e_1 terminates in n steps with value v and e_2 terminates in m steps with value w then $n + m$ steps into the future, we can update our resources to satisfy $\Phi(v, w)$ while being able to open all invariants in $\mathcal{E}$ atomically during every step. Note that this is a *termination-insensitive* relation; we assume both relations terminate and then the postcondition should hold. This is in contrast to the earlier relational models in Iris which have been *termination-sensitive* and definable using the standard weakest preconditions of Iris. Moreover, notice that we count the steps taken on *both* sides of the relation by including later modalities for both executions—Rajani and Garg [2020] and earlier relational models in Iris only count steps for one of the programs.

Formally, we define this predicate using two modal weakest precondition instances where the second is defined in terms of the first. We use this approach rather than defining the binary predicate directly as it will allow us to re-use the proof rules for modal weakest preconditions to reason about the individual programs when arguing binary relatedness. The definitions are somewhat technical and can easily be skipped on a first reading.

Let $\mathcal{M}_I$ be defined by

$$\mathsf{M}^m_{\mathcal{E};n}(\Phi) \triangleq \left({}^{\mathcal{E}}\Rrightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rrightarrow^{\mathcal{E}}\right)^{n+m} \Rrightarrow_{\mathcal{E}} \Phi()$$
$$\mathsf{BindCond}(n, m, f, g) \triangleq m \leq n \wedge \forall x, f(x) = m - n \wedge \lambda\_, g = id.$$

The modality's index is a natural number m that adds m extra logical steps to the step-taking update modality and the postcondition parameter is unit. The bind condition ensures that the logical steps "add up" and that the post condition is otherwise unmodified. The modality defined by $\mathcal{M}_I$ is valid.

Now, let $\mathcal{M}_{\times\Rrightarrow}$ be defined by

$$\mathsf{M}^e_{\mathcal{E};n}(\Phi) \triangleq \mathsf{mwp}^{\mathcal{M}_I;n}_{\mathcal{E}}\, e\, \{w, m.\ \Phi(w, m)\}$$
$$\mathsf{BindCond}(e_1, e_2, f, g) \triangleq \exists K.\, e_1 = K[e_2] \wedge\ g = \lambda(v_1, n_1), (v_2, n_2).(v_2, n_1 + n_2) \wedge$$
$$\forall v, k.\, f(v, k) = K[v].$$

The modality is the MWP connective instantiated with $\mathcal{M}_I$. The bind condition reflects the preconditions for the binding lemma for the inner connective and that the steps taken are propagated to the postcondition. The modality defined by $\mathcal{M}_{\times\Rrightarrow}$ is valid. We now define $\mathsf{mwp}_{\mathcal{E}}\, e \sim e'\, \{v, w.\ \Phi\}$ to be $\mathsf{mwp}^{\mathcal{M}_{\times\Rrightarrow}, e'}_{\mathcal{E}}\, e\, \{v, \_, (w, \_).\ \Phi\}$; by unfolding the definitions one can see that it indeed satisfies the desired relational predicate in Equation (4).

A crucial property of the binary relation defined in the above example is the following *binary* version of the bind rule, which intuitively means that we can do *relational* reasoning in a local way.

Lemma 3.6 (Binary step-taking update MWP - bind).

$$\frac{\mathsf{mwp}\ e \sim e' \left\{v, v'.\ \mathsf{mwp}\ K[v] \sim K'[v']\ \{\Phi\}\right\}}{\mathsf{mwp}\ K[e] \sim K'[e']\ \{\Phi\}}$$

At the same time, essential for our logical-relations model, we have all the proof rules for reasoning about the two computations individually. This is embodied in Lemma 3.7 that allows us to reason about each computation using the unary modal weakest precondition instance from Example 3.2.

Lemma 3.7 (Unary-binary step-taking update MWP).

$$\mathsf{mwp}^{\mathcal{M}_{\Rrightarrow}}_{\mathcal{E}}\, e_1 \left\{v.\ \mathsf{mwp}^{\mathcal{M}_{\Rrightarrow}}_{\mathcal{E}}\, e_2\, \{w.\ \Phi(v, w)\}\right\} \twoheadrightarrow \mathsf{mwp}\ e_1 \sim e_2\ \{\Phi\}$$
$$\mathsf{mwp}^{\mathcal{M}_{\Rrightarrow}}_{\mathcal{E}}\, e_2 \left\{w.\ \mathsf{mwp}^{\mathcal{M}_{\Rrightarrow}}_{\mathcal{E}}\, e_1\, \{v.\ \Phi(v, w)\}\right\} \twoheadrightarrow \mathsf{mwp}\ e_1 \sim e_2\ \{\Phi\}$$

Recall that the modal weakest precondition connective is defined as a proposition in Iris of type *iProp*. To demonstrate that the theory actually makes the expected statements about program execution *in the meta logic*, once and for all, for *any* language and for *any* modality, we prove a general adequacy theorem for the modal weakest precondition theory. The details of this general theorem is relegated to the Coq formalization. For concrete languages and modalities, specific adequacy theorems such as the following hold as simple corollaries.

Theorem 3.8 (Adequacy of binary step-taking update MWP with λ_{sec}). *Let φ be a first-order (meta-logic) predicate over values. Suppose $\mathsf{mwp}_{\mathcal{E}}\, e_1 \sim e_2\, \{\varphi\}$ is derivable. If $(\sigma_1, e_1) \to^* (\sigma_1', v_1)$ and $(\sigma_2, e_2) \to^* (\sigma_2', v_2)$ then $\varphi(v_1, v_2)$ holds at the meta-level.*

3.3 Expression Relations

We now return to the expression relations of our logical-relations model, which are defined using modal weakest preconditions; see Figure 6 and Figure 4.

The binary interpretation relates expressions at τ that only terminate with related values at τ. This is defined directly using the binary connective derived in Example 3.5.

$$\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e, e') \triangleq \mathsf{mwp}\ e_1 \sim e_2 \left\{[\![\tau]\!]^{\rho}_{\Theta}\right\}.$$

Recall that the unary interpretation is intended to be inhabited by terms that have no ζ-observable side-effects. We observe that only expressions in high-labeled contexts (where control flow depends on high-labeled data) are critical; low-labeled contexts are ζ-observable and should not be considered. To incorporate this observation, the unary expression relation is annotated with a pc label. The unary value interpretation of the function type and the polymorphic types pass on the latent effect label as this parameter. If pc is a high label ($[\![pc]\!]_{\rho} \not\sqsubseteq \zeta$), then e is in the expression interpretation of τ if e satisfies the unary modal weakest precondition from Example 3.2.

$$\mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e) \triangleq [\![pc]\!]_{\rho} \not\sqsubseteq \zeta \Rightarrow \mathsf{mwp}^{\mathcal{M}_{\Rrightarrow}}\ e \left\{[\![\tau]\!]^{\rho}_{\Delta}\right\}.$$

With the value and expression relations for closed values and expressions defined, logical relatedness for open terms is now defined by closing them by related substitutions, as is usual for logical relations. Substitutions are related using the environment relation interpretations denoted $\mathcal{G}$ in Figure 6 and Figure 4.

The *unary semantic typing judgment* (logical relation) is defined as

$$\Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau \triangleq \square \left(\begin{array}{l} \forall \Delta, \rho, \vec{v}\,.\, \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Delta) * \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(\rho) \mathrel{-\!\!*} \\ \qquad \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Delta}(\vec{v}) \mathrel{-\!\!*} \mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e[\vec{v}/\vec{x}]) \end{array} \right)$$

and the *binary semantic typing judgment* as

$$\Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e' : \tau \triangleq \square \left(\begin{array}{l} \forall \Theta, \rho, \vec{v}, \overrightarrow{v'}\,.\, \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Theta) * \mathrm{dom}(\Psi) \subseteq \mathrm{dom}(\rho) \mathrel{-\!\!*} \\ \qquad \mathit{Coh}(\Theta) * \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v}, \overrightarrow{v'}) \mathrel{-\!\!*} \mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e[\vec{v}/\vec{x}], e'[\overrightarrow{v'}/\vec{x}]) \end{array} \right).$$

Notice that the binary judgment additionally requires the semantic type environment to be coherent.

4 THE FUNDAMENTAL THEOREMS AND SOUNDNESS

It is straightforward to show the unary fundamental theorem by structural induction on the typing derivation. All proofs are carried out at an abstraction level similar to the structural rules shown in this paper. This is enabled by our formulation of the modal weakest precondition theory and the MoSeL framework [Krebbers et al. 2018] for manipulating the Iris connectives.

Theorem 4.1 (Unary fundamental theorem).

$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \Rightarrow \Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau$$

Similarly, the binary fundamental theorem also follows by structural induction in the typing derivation and the structural rules of the binary modal weakest precondition and its interaction with the unary modal weakest precondition. However, it also relies heavily on the binary-unary subsumption property which is the content of the following lemma.

Lemma 4.2 (Binary-unary subsumption).

$$\mathit{Coh}(\Theta) * [\![\tau]\!]^{\rho}_{\Theta}(v, v') \mathrel{-\!\!*} [\![\tau]\!]^{\rho}_{\Theta_L}(v) * [\![\tau]\!]^{\rho}_{\Theta_R}(v').$$

To see why this property is crucial and to exemplify how the binary and unary relations interact, consider the compatibility lemma for conditional expressions. This lemma concludes

$$\Xi \mid \Psi \mid \Gamma \vDash \text{if } e \text{ then } e_1 \text{ else } e_2 \approx_\zeta \text{if } e' \text{ then } e_1' \text{ else } e_2' : \tau$$

given well-typed sub-terms and $\Xi \mid \Psi \mid \Gamma \vDash e \approx_\zeta e' : \mathbb{B}^\ell$, $\Xi \mid \Psi \mid \Gamma \vDash e_i \approx_\zeta e_i' : \tau$, and $\tau \searrow \ell$, *cf.*, T-IF for conditional expressions in Figure 1.

Unfolding the definition of the binary semantic typing judgment, this means that given related substitutions $\vec{v}$ and $\overrightarrow{v'}$, *i.e.*, $\mathcal{G}[\![\Gamma]\!]^\rho_\Theta(\vec{v}, \overrightarrow{v'})$, and $Coh(\Theta)$ it suffices to show

$$\mathcal{E}[\![\tau]\!]^\rho_\Theta(\text{if } e[\vec{v}/\vec{x}] \text{ then } e_1[\vec{v}/\vec{x}] \text{ else } e_2[\vec{v}/\vec{x}], \text{if } e'[\overrightarrow{v'}/\vec{x}] \text{ then } e_1'[\overrightarrow{v'}/\vec{x}] \text{ else } e_2'[\overrightarrow{v'}/\vec{x}]).$$

The proof continues by considering whether label ℓ of the branch condition is ζ-observable or not, *i.e.*, whether $\ell \sqsubseteq \zeta$ or $\ell \not\sqsubseteq \zeta$. In the case where the label is *not* observable this means that given $\tau = t^{\ell'}$ then $\ell' \not\sqsubseteq \zeta$ as well and hence the values that e and e' evaluate to are (potentially) different, *cf.*, the binary value interpretation of labeled Booleans. In turn, this means evaluation of the two conditional expressions might continue through different branches, *i.e.*, we end up having to show $\mathcal{E}[\![\tau]\!]^\rho_\Theta(e_1[\vec{v}/\vec{x}], e_2'[\overrightarrow{v'}/\vec{x}])$ and $\mathcal{E}[\![\tau]\!]^\rho_\Theta(e_2[\vec{v}/\vec{x}], e_1'[\overrightarrow{v'}/\vec{x}])$.

Using Lemma 3.7 we can reason about the two expressions individually, and the statements follow from the unary fundamental theorem (Theorem 4.1). However, the assumption $\mathcal{G}[\![\Gamma]\!]^\rho_\Theta(\vec{v}, \overrightarrow{v'})$ on substitutions $\vec{v}$ and $\overrightarrow{v'}$ is *binary*—in order to use the unary fundamental theorem, the related substitutions individually need to satisfy the unary environment interpretations, *i.e.*, $\mathcal{G}[\![\Gamma]\!]^\rho_{\Theta_L}(\vec{v})$ and $\mathcal{G}[\![\Gamma]\!]^\rho_{\Theta_R}(\overrightarrow{v'})$. This fact follows from Lemma 4.2.

Theorem 4.3 (Binary fundamental theorem).

$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \Rightarrow \Xi \mid \Psi \mid \Gamma \vDash e \approx_\zeta e : \tau.$$

By composing the binary fundamental theorem and the adequacy theorem for the binary modal weakest precondition instance (Theorem 3.8) we show our final soundness theorem, which shows that our type system does indeed imply termination-insensitive noninterference.

Theorem 4.4 (Termination-Insensitive Noninterference). *Let ζ, $\top$ and $\bot$ be labels drawn from a join-semilattice such that $\bot \sqsubseteq \zeta$ and $\top \not\sqsubseteq \zeta$. If*

$$\cdot \mid \cdot \mid x : \mathbb{B}^\top \vdash_\bot e : \mathbb{B}^\bot,$$
$$\cdot \mid \cdot \mid \cdot \vdash_\bot v_1 : \mathbb{B}^\top, \text{ and } \cdot \mid \cdot \mid \cdot \vdash_\bot v_2 : \mathbb{B}^\top$$

then

$$(\emptyset, e[v_1/x]) \to^* (\sigma_1, v_1') \wedge (\emptyset, e[v_2/x]) \to^* (\sigma_2, v_2') \Rightarrow v_1' = v_2'.$$

5 EXAMPLES OF SEMANTIC TYPING

By the soundness theorem (Theorem 4.4) above we now know that any syntactically well-typed program satisfies noninterference. Our model also allows us to *semantically* type programs that are not syntactically well-typed but are nevertheless secure, for reasons too subtle for the syntactic type system to discover. Semantically well-typed programs can then be safely composed with *syntactically* well-typed programs while maintaining noninterference. To see how this works in practice, we will first examine a few small programs that are safe to execute but untypable in our static type system. Later, we will move on to more intricate examples and show how we can prove that these are secure and therefore also safe to compose with other syntactically typed programs. The examples in this section thus illustrate some of the strengths of our semantic approach to noninterference. The proofs of the examples rely both on our semantic model of types (in particular

abstract types) and also on our ability to use Iris ghost state and Iris invariants to reason about intricate invariants on local state. For reasons of space, however, proofs have been omitted from the paper; they can be found in the accompanying Coq formalization.

In the following examples we will often omit labels on composite types to simplify the presentation. An omitted label should always be read as being label $\bot$.

To start off, consider the trivial program $\lambda\, v.\, v * 0$ that multiples its input by zero. Syntactically, it cannot be typed at $\mathbb{N}^\top \to \mathbb{N}^\bot$ as its output seemingly depends on its input which is at a high security label. But, by simple arithmetic, the output is always constant and the function can thus be shown to be in the semantic interpretation $[\![\mathbb{N}^\top \to \mathbb{N}^\bot]\!]$ of the type. Hence it can be safely composed with any syntactically well-typed code that relies on a function of this type.

Next, consider programs (5) and (6)

$$\mathsf{let}\, x = !\, l\, \mathsf{in}\, l \leftarrow !\, h;\ \ldots;\ l \leftarrow x \tag{5}$$

$$(\mathsf{if}\ !\, h = 42\ \mathsf{then}\, l \leftarrow 0\, \mathsf{else}\, l \leftarrow 1);\ l \leftarrow 0 \tag{6}$$

which both *temporarily* store information (both explicitly and implicitly) from a sensitive reference h into a public reference l. Due to the flow-insensitive nature of the syntactic type system, both of the programs cannot be type checked, as sensitive information is not allowed to flow into a public reference. However, by restoring public information in the reference, both programs are in fact secure. In both cases, location l inhabits type $\mathsf{ref}\,(\mathbb{N}^\bot)^\bot$ which, *cf.* Figure 4, means that its contents must invariantly be binary related at $\mathbb{N}^\bot$. To prove that these examples are semantically well-typed, it is necessary to keep the invariant open for the full execution of the program. Recall that the modal weakest precondition (Example 3.5) used to define the binary expression relation only allows invariants to be opened atomically during every step, so it seems that it might be difficult to show semantic well-typedness. But, fortunately, we can prove semantic well-typedness of these examples using a binary version of the modal weakest precondition instance from Example 3.1 that allows invariants to stay open for the full execution of the program.

Lemma 5.1 (Binary update MWP implies binary step-update MWP). *If either e_1 or e_2 are able to make progress then*

$$\left(\mathcal{E}_1 \Rrightarrow^{\mathcal{E}_2} \triangleright\, \mathsf{mwp}^{\mathcal{M}_{\times \Rrightarrow}; e_2}_{\mathcal{E}_2}\, e_1\, \{v, n, b.\ \mathcal{E}_2 \Rrightarrow^{\mathcal{E}_1} \Phi(v, n, b)\}\right) \mathrel{-\!\!*} \mathsf{mwp}_{\mathcal{E}_1}\, e_1 \sim e_2\, \{\Phi\}.$$

For space reasons, we relegate the details to the appendix and the Coq formalization, however, we emphasize that this example illustrates the generality and flexibility offered by our modal weakest precondition theory.

5.1 Static Semantic Typing Instead of Dynamic Enforcement

We now consider an example adapted from Fennell and Thiemann [2013], namely a report processing application containing security-typed operations that process reports by reference. The example contains code fragments that the type system of Fennell and Thiemann [2013] cannot statically type check. Instead, the authors of *loc. cit.* propose to use a gradual security type system where security levels are checked at run-time. Those code fragments cannot be type checked by our syntactic type system either, but we *can* prove that they are semantically well-typed. Not only does this prove the example secure, it also avoids unnecessary run-time cost while still allowing the code to be composed with the remainder of the syntactically well-typed report processing application.

The basic operations of the report processing application include

$$\mathit{sendToManager} : \mathsf{ref}\,(\mathit{Report}^\top) \xrightarrow{\top} 1$$

$$\mathit{sendToFacebook} : \mathsf{ref}\,(\mathit{Report}^\bot) \xrightarrow{\bot} 1$$

where the idea is that *sendToManager* can process sensitive reports and send those to trusted managers (by assigning to a reference, *cf.* the $\top$ latent effect label) whereas *sendToFacebook* can only process public reports and thus has a $\bot$ latent effect label.

Now, Fennell and Thiemann consider an extension of the application with a utility function *addPrivileged*, which adds privileged information to a report before passing it to a $worker$ (like one of the basic operations in the above):

$$\begin{aligned} \mathit{addPrivileged} \triangleq \lambda\, \mathit{isPrivileged}, \mathit{worker}, \mathit{report}.\ & \\ & \mathsf{if}\ \mathit{isPrivileged}\ \mathsf{then}\ \mathit{report} \leftarrow !\,\mathit{report} + !\,h\ \mathsf{else}\ () \\ & \mathit{worker}\ \mathit{report} \end{aligned}$$

The flag $isPrivileged$ indicates whether the $worker$ argument has a sufficient security level to handle a privileged report. If the flag is true, sensitive information is retrieved from a global reference h and appended to the report. Otherwise, the $worker$ is invoked with an unmodified report. Both *addPrivileged* itself and the application *addPrivileged* true *sendToManager* syntactically type checks, the former at the type $\mathsf{ref}(\mathit{Report}^{\top}) \xrightarrow{\top} 1$, as *sendToManager* can safely operate on sensitive information. However, the code fragment *addPrivileged* false *sendToFacebook* does *not* type check, even though it is safe and no sensitive information is leaked. The code does not type check because the type system does not track the dependency between the $isPrivileged$ flag and the $worker$'s security clearance. Using our model, however, we can prove that it can be semantically typed at $\mathsf{ref}(\mathit{Report}^{\bot}) \xrightarrow{\bot} 1$, meaning that the code can be composed with other syntactically well-typed report operations, without introducing any runtime labels.

Proposition 5.2. *Let* $\mathit{addPFB} \triangleq \mathit{addPrivileged}\ \mathsf{false}\ \mathit{sendToFacebook}$ *then*

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{addPFB} \approx_{\zeta} \mathit{addPFB} : \mathsf{ref}(\mathit{Report}^{\bot}) \xrightarrow{\bot} 1^{\bot}$$

The proof is straightforward and follows by symbolic execution of the program.

5.2 Value-Dependent Classification and Modularity

Traditionally, information-flow control systems partition the heap into compartments for each security level. This is impractical for realistic settings where resources, such as the heap, can be shared. To address this issue, some recent information-flow systems [Gregersen et al. 2019; Lourenço and Caires 2015; Murray et al. 2016; Nanevski et al. 2011; Zheng and Myers 2007] support *value-dependent classification* policies. These policies describe a relationship between two values, such that the value of one decides the classification-level of the other. We now demonstrate that our semantic model supports reasoning about value-dependent classification policies; we also use this example to show an application of existential types to increase modularity by hiding the value dependency.

Consider the example of a program with value-dependent classification below.

$$\begin{aligned} \mathit{valDep} \triangleq \lambda\, f.\ & \mathsf{let}\ \mathit{dep} = \mathsf{ref}(\mathsf{true}, \mathit{secret})\ \mathsf{in} \\ & f\ \mathit{dep}; \\ & \mathsf{let}\ \mathit{tmp} = !\,\mathit{dep}\ \mathsf{in} \\ & \mathsf{if}\ \pi_1\ \mathit{tmp}\ \ \mathsf{then}\ 42\ \mathsf{else}\ \pi_2\ \mathit{tmp} \end{aligned}$$

The program allocates a reference dep which points to a pair consisting of a Boolean and a number. If the Boolean is true, the contents of the second component should be regarded as secret; otherwise public. The reference is passed to the function f which therefore must uphold this invariant for the program to be secure. Finally, the contents of dep is inspected and if the Boolean is true (*i.e.*,

the content is secret), we ignore the second component and return 42 and otherwise it is safe to return the second component. Ideally, we would like to show that given a function f, the pair $(\mathit{valDep}\ f, \mathit{valDep}\ f)$ is in the binary interpretation $[\![\mathbb{N}^{\bot}]\!]$. Obviously this does not hold for an arbitrary function f; to prove it we need to know that f maintains the following invariant on dep:

$$\exists b, d_L, d_R. dep_L \mapsto_L (b, d_L) * dep_R \mapsto_R (b, d_R) * [\![\mathbb{N}^{\mathsf{if}\ b\ \mathsf{then}\ \top\ \mathsf{else}\ \bot}]\!](d_L, d_R)$$

The invariant ensures that f cannot write a secret to dep without also setting the Boolean to true.

This example shows how we can encode value-dependent classifications in our system but with the cost of burdening the client of the above program with showing that f upholds the invariant. The issue is that the client's code gets direct access to the reference with the classification, but the static type system is oblivious to the semantic meaning of it.

To alleviate this problem, we can instead hide the reference in an existential package. This allows us to only expose accessor- and mutator methods to the client, such that the client only needs to statically type check against these methods. The code for this variant is seen below.

$$\begin{aligned}
\mathit{valDepPack} \triangleq\ & \mathsf{let}\ get = \lambda\, dep.\ \mathsf{let}\ c = !\, dep\ \mathsf{in}\ \mathsf{if}\ \pi_1\, c\ \mathsf{then}\ \mathsf{inj}_1\ (\pi_2\, c)\ \mathsf{else}\ \mathsf{inj}_2\ (\pi_2\, c)\ \mathsf{in} \\
& \mathsf{let}\ setL = \lambda\, dep, v.\ dep \leftarrow (\mathsf{false}, v)\ \mathsf{in} \\
& \mathsf{let}\ setH = \lambda\, dep, v.\ dep \leftarrow (\mathsf{true}, v)\ \mathsf{in} \\
& \mathsf{pack}\ (\mathsf{ref}(\mathsf{true}, secret), get, setL, setH)
\end{aligned}$$

Using our semantic model, we can prove that this program inhabits the interpretation of an existential type.

Proposition 5.3.

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{valDepPack} \approx_{\zeta} \mathit{valDepPack} : \exists \alpha. \left(\alpha^{\bot} \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\top} + \mathbb{N}^{\bot}\right) \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\bot} \xrightarrow{\bot} 1\right) \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\top} \xrightarrow{\bot} 1\right)\right)$$

This allows statically typed clients to store both secret and public information in the reference, but they must do so through the mutators $setL$ and $setH$. When clients want to read the reference, they can do so with the get function which gives a value of type $\mathbb{N}^{\top} + \mathbb{N}^{\bot}$.

5.3 Computing with Memoization

The following example shows an implementation of a service for computing a function with memoization. The service takes a function f as input and then allows clients to compute f on client-provided inputs; when doing so, the service remembers the last input and corresponding result and returns this directly if the client asks for the same input again. The idea, of course, would be that the function f is very expensive to compute, so the client would therefore like to memoize the already computed values in case they are needed again. This behavior is implemented with a single reference that points to a tuple consisting of the last input value and the corresponding result. The code for this service is shown below.

$$\begin{aligned}
\mathit{memoize} \triangleq\ & \lambda\, f, \mathit{init}. \\
& \mathsf{let}\ \mathit{cache} = \mathsf{ref}(\mathit{init}, f\ \mathit{init})\ \mathsf{in} \\
& \mathsf{let}\ \mathit{recompute} = \lambda\, v.\ \mathsf{let}\ \mathit{result} = f\ v\ \mathsf{in}\ \mathit{cache} \leftarrow (v, \mathit{result}); \mathit{result}\ \mathsf{in} \\
& \lambda\, v.\ \mathsf{let}\ (w, \mathit{result}) = !\, \mathit{cache}\ \mathsf{in} \\
& \quad \mathsf{if}\ v = w\ \mathsf{then}\ \mathit{result}\ \mathsf{else}\ \mathit{recompute}\ v
\end{aligned}$$

First, let us see why we cannot give a static type to this program. Suppose we have a function f of type $\mathbb{N}^{\bot} \xrightarrow{\top} \mathbb{N}^{\bot}$. The issue then is giving a reasonable type to the reference for the cache. If we type it at $\bot$, then the returned function will necessarily have a latent effect label at $\bot$, and it is therefore not interchangeable with the input function. If we instead type the reference at $\top$, then we must label the output of the resulting function to $\top$ as well.

Clearly, we cannot hope to give a reasonable type to this program using our static type system, so we will instead try to define a security condition for it. For a suitable function f from $\mathbb{N}^{\ell}$ to $\mathbb{N}^{\ell}$, we would like to show that *memoize* $f\ 0$ has the semantic type $[\![\mathbb{N}^{\ell} \xrightarrow{\top} \mathbb{N}^{\ell}]\!]$, so any well-typed client can use this to compute f with caching.

Note that the latent effect label of the returned function is $\top$ even though the function writes to the cache. The secrecy of the cache itself is independent of the secrecy of the outputs of the function f, but instead varies based on the secrecy of the context the last call that updated the cache happened in.

For this to be secure, *memoize* relies on the input function f to "act" purely. Intuitively, the function must behave as if it was a pure function on all terminating inputs. This rules out programs such as the following that tries to exploit the memoization by counting the number of times f has been called:

$$\begin{array}{l}\mathsf{let}\ \mathit{counter} = \mathsf{ref}(0)\ \mathsf{in}\\ \mathsf{let}\ f' = \mathit{memoize}\ (\lambda\ \_.\ \mathit{counter} \leftarrow (!\ \mathit{counter} + 1); !\ \mathit{counter})\ 0\ \mathsf{in}\\ \mathsf{if}\ \mathit{secret}\ \mathsf{then}\ f'\ 0\ \mathsf{else}\ ();\\ f'\ 0\end{array}$$

This allows us to prove that *memoize* $f\ 0$ is semantically secure and we can therefore link this with any piece of statically typed code that makes use of this function with memoization, while maintaining security of the whole program. We refer to the Coq formalization for more details.

Proposition 5.4. *For any purely acting function f from $\mathbb{N}^{\ell}$ to $\mathbb{N}^{\ell}$, we have that*

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{memoize}\ f\ 0 \approx_{\zeta} \mathit{memoize}\ f\ 0 : \mathbb{N}^{\ell} \xrightarrow{\top} \mathbb{N}^{\ell}$$

5.4 Higher-order Functions and Dynamicly Allocated References

Consider the following variation by Frumin et al. [2019] of the "awkward" example, originally given by Pitts and Stark [1998] when studying the challenges of proving contextual equivalence about higher-order functions and state:

$$\mathit{awk} \triangleq \lambda\, v.\, \mathsf{let}\, x = \mathsf{ref}(v)\, \mathsf{in}\, \lambda\, f.\, x \leftarrow 1; f(); !\, x$$

When applied to a value v, the program returns a closure that, when invoked with a function f, always returns the constant value 1. From an information-flow control perspective this means that even if *awk* is invoked with a sensitive argument, it will always be safe to consider the output of the closure as public. This fact crucially relies on the reference x being local to the closure. The program is not well-typed using the syntactic type system as x has to contain both sensitive and public values. However, we can semantically type *awk*.

Proposition 5.5. $\cdot \mid \cdot \mid \cdot \vDash \mathit{awk} \approx_{\zeta} \mathit{awk} : \mathbb{N}^{\top} \xrightarrow{\bot} (1 \xrightarrow{\bot} 1) \xrightarrow{\bot} \mathbb{N}^{\bot}$

To prove that the contents of the reference is in fact public after invoking the function, we use an invariant with a two-state protocol (defined using Iris ghost state) on the contents of the reference; see the accompanying Coq code for more details.

5.5 Parametricity and Free Theorems

We can use our model to prove free theorems; here are two simple examples. As far as we know, such properties have not been shown for information-flow control type systems before.

Proposition 5.6. *If* $\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \forall_{\ell_1}\, \alpha.\, \alpha^{\ell_2} \xrightarrow{\ell_3} \alpha^{\ell_2}$ *and* $(\sigma, e\ \_\ v) \to^* (\sigma', v')$ *then* $v = v'$.

Proposition 5.7. *There does not exist a non-diverging* e *where* $\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \forall_{\top}\, \alpha.\, \alpha^{\top} \xrightarrow{\top} \alpha^{\bot}$.

6 RELATED WORK

In the following, we consider related work that has not been treated already throughout the paper.

Logical relations for information-flow security. Sabelfeld and Sands [1999, 2001] present a model of information-flow security based on partial-equivalence relations; they establish various semantic properties about the model and use it to prove a termination-sensitive notion of noninterference for a calculus equipped with a simple security type system, first-order state, and probabilistic choice. Zdancewic [2002] proves a security-typed simply-typed lambda calculus sound using a logical-relations argument but uses a translation-based argument when considering mutable state. Abadi et al. [1999] introduce *the dependency core calculus* (DCC), a pure calculus designed to capture the central notion of dependency arising in a setting like information-flow security. They prove noninterference using a denotational semantics based on partial equivalence relations. Heintze and Riecke [1998] also prove noninterference of the pure fragment of the SLam calculus using logical relations. Pottier and Conchon [2000] conjecture that the noninterference proofs of Abadi et al. [1999]; Heintze and Riecke [1998] cannot easily deal with recursive or polymorphic types. Compared to our work, all of the above consider simpler settings with respect to language features and type systems. Using Iris, Frumin et al. [2019] present a separation logic for proving a notion of timing-sensitive noninterference of concurrent programs. On top of this logic, they build a logical-relations model of a simple type system that allows them to compositionally verify and integrate syntactically well-typed and ill-typed parts. In contrast to *loc. cit.* we focus on termination-insensitive noninterference and (in part for this reason) our type system is more expressive.

Our models are directly inspired by Rajani and Garg [2020] that describe a step-indexed Kripke-style logical-relations model for two information-flow control type systems for a sequential language with higher-order state similar to ours. However, their type system does not support impredicative polymorphism and their semantic model cannot easily be extended to support this due to their use of syntactic worlds. Our semantic handling of label polymorphism is also different due to our use of semantic worlds. Rajani and Garg use their relation to prove that the fine-grained and coarse-grained static IFC systems are equivalent; Vassena et al. [2019] show a similar result for dynamic information-flow control systems.

Noninterference and polymorphism. Abadi [2006] introduces a polymorphic DCC in the style of System F for access control in distributed systems. Inspired by the polymorphic DCC, Arden and Myers [2016] study a pure authorization calculus with polymorphic type-abstraction. Pottier and Simonet [2003] study an ML-like language with let-polymorphism, recursion, references, and exceptions. In contrast to our work, these works consider less expressive notions of polymorphism than us or study pure calculi and prove noninterference using a syntactic approach which does not scale to relational reasoning for impredicative polymorphism in the presence of higher-order state. Morever, they do not benefit from the semantic approach with compositional integration of syntactically well-typed and syntactically ill-typed components.

The proof technique for noninterference of DCC by Abadi et al. [1999] suggests that it is possible to use the parametric polymorphism in System F to model the dependency of DCC. Based on

previous work of Tse and Zdancewic [2004], Bowman and Ahmed [2015] provide a translation from the recursion-free fragment of DCC to F_ω, translating noninterference into parametricity. Algehed and Bernardy [2019] leverage parametricity of the Calculus of Constructions to prove noninterference for a polyvariant variation of DCC and Algehed et al. [2020] show noninterference of a dynamic information-flow control library using a parametricity theorem.

All these works *model* information-flow properties using parametricity whereas we add impredicative type polymorphism to a security-typed language.

7 CONCLUSION

We present the first semantic model of an information-flow control type system with impredicative polymorphism (universal and existential types), recursive types, and general reference types, and show how we can use our model to reason about syntactically ill-typed but semantically sound code. We showcase our methodology on multiple interesting examples and how our approach allows for compositional integration. Our semantic model guarantees termination-insensitive noninterference and we formalize it using logical relations on top of the Iris program logic framework. To do so, we introduce a novel re-usable program logic construct and theory of Modal Weakest Preconditions.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project.

REFERENCES

Martín Abadi. 2006. Access control in a core calculus of dependency. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. 263–273. https://doi.org/10.1145/1159803.1159839

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 147–160. https://doi.org/10.1145/292540.292555

Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References A Stratified Semantics of General References. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 75. https://doi.org/10.1109/LICS.2002.1029818

Maximilian Algehed and Jean-Philippe Bernardy. 2019. Simple noninterference from parametricity. *Proc. ACM Program. Lang.* 3, ICFP (2019), 89:1–89:22. https://doi.org/10.1145/3341693

Maximilian Algehed, Jean-Philippe Bernardy, and Catalin Hritcu. 2020. Dynamic IFC Theorems for Free! *CoRR* abs/2005.04722 (2020). arXiv:2005.04722 https://arxiv.org/abs/2005.04722

Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*. 77–89. https://doi.org/10.1145/3139337.3139338

Owen Arden and Andrew C. Myers. 2016. A Calculus for Flow-Limited Authorization. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 135–149. https://doi.org/10.1109/CSF.2016.17

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf. (2017).

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed kripke models over recursive worlds. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 119–132. https://doi.org/10.1145/1926385.1926401

William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 101–113. https://doi.org/10.1145/2784731.2784733

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. 71–80. https://doi.org/10.1109/LICS.2009.34

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. 224–239. https://doi.org/10.1109/CSF.2013.22

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2019. Compositional Non-Interference for Fine-Grained Concurrent Programs. *CoRR* abs/1910.00905 (2019). arXiv:1910.00905 http://arxiv.org/abs/1910.00905

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. *1982 IEEE Symposium on Security and Privacy* (Apr 1982). https://doi.org/10.1109/sp.1982.10014

Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. 2019. A Dependently Typed Library for Static Information-Flow Control in Idris. In *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. 51–75. https://doi.org/10.1007/978-3-030-17138-4_3

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 365–377. https://doi.org/10.1145/268946.268976

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

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 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. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

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 *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 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 *Principles of Programming Languages (POPL)*.

Peng Li and Steve Zdancewic. 2006. Encoding Information Flow in Haskell. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. 16. https://doi.org/10.1109/CSFW.2006.13

Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 317–328. https://doi.org/10.1145/2676726.2676994

Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. 2016. Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 417–431. https://doi.org/10.1109/CSF.2016.36

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 228–241. https://doi.org/10.1145/292540.292561

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. 165–179. https://doi.org/10.1109/SP.2011.12

Andrew M. Pitts and Ian D. B. Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts (Eds.). Cambridge University Press, 227–273.

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 361–375. https://doi.org/10.1007/BFb0037118

François Pottier and Sylvain Conchon. 2000. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. 46–57. https://doi.org/10.1145/351240.351245

François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (2003), 117–158. https://doi.org/10.1145/596980.596983

Vineet Rajani and Deepak Garg. 2020. On the expressiveness and semantics of information flow types. *Journal of Computer Security* 28, 1 (2020), 129–156. https://doi.org/10.3233/JCS-191382

Alejandro Russo. 2015. Functional pearl: two can keep a secret, if one of them uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 280–288. https://doi.org/10.1145/2784731.2784756

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A library for light-weight information-flow security in haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 13–24. https://doi.org/10.1145/1411286.1411289

Andrei Sabelfeld and David Sands. 1999. A PER Model of Secure Information Flow in Sequential Programs. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. 40–58. https://doi.org/10.1007/3-540-49099-X_4

Andrei Sabelfeld and David Sands. 2001. A Per Model of Secure Information Flow in Sequential Programs. *High. Order Symb. Comput.* 14, 1 (2001), 59–91. https://doi.org/10.1023/A:1011553200337

Vincent Simonet. 2003a. Flow Caml in a Nutshell. In *Proc. of the first APPSEM-II workshop*, Graham Hutton (Ed.). Nottingham, United Kingdom.

Vincnet Simonet. 2003b. *The Flow Caml system.* http://cristal.inria.fr/~simonet/soft/flowcaml

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 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. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. 115–125. https://doi.org/10.1145/1016850.1016868

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. https://doi.org/10.1145/2500365.2500600

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC A verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming* 95 (2018), 148 – 180. https://doi.org/10.1016/j.jlamp.2017.12.003

Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine- to coarse-grained dynamic information flow control and back. *Proc. ACM Program. Lang.* 3, POPL (2019), 76:1–76:31. https://doi.org/10.1145/3290389

Stephan Arthur Zdancewic. 2002. *Programming Languages for Information Security.* Ph.D. Dissertation. Cornell University, USA.

Lantian Zheng and Andrew C. Myers. 2007. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.* 6, 2-3 (2007), 67–84. https://doi.org/10.1007/s10207-007-0019-9