Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, Effect Safety, and Equational Theory

YUYAN BAO, Augusta University, USA
SONGLIN JIA, Purdue University, USA
GUANNAN WEI, Tufts University, USA
OLIVER BRAČEVAC, EPFL, Switzerland
TIARK ROMPF, Purdue University, USA

Reachability types are a recent proposal to bring Rust-style reasoning about memory properties to higher-level languages, with a focus on higher-order functions, parametric types, and shared mutable state – features that are only partially supported by current techniques as employed in Rust. While prior work has established key type soundness results for reachability types using the usual syntactic techniques of progress and preservation, stronger metatheoretic properties have so far been unexplored. This paper presents an alternative semantic model of reachability types using logical relations, providing a framework in which we study key properties of interest: (1) semantic type soundness, including of not syntactically well-typed code fragments, (2) termination, especially in the presence of higher-order mutable references, (3) effect safety, especially the absence of observable mutation, and, finally, (4) program equivalence, especially reordering of non-interfering expressions for parallelization or compiler optimization.

CCS Concepts: • **Software and its engineering → Functional languages**; **General programming languages**; **Semantics**; • **Theory of computation → Program reasoning**.

Additional Key Words and Phrases: type systems, reachability types, effect systems, logical relations, equational reasoning

ACM Reference Format:
Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, Effect Safety, and Equational Theory. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 338 (October 2025), 28 pages. https://doi.org/10.1145/3763116

1 Introduction

Reachability types are a recent proposal to bring Rust-style reasoning about memory properties to higher-level languages. Instead of Rust's rigid "shared XOR mutable" model, reachability types emphasize *tracking* of resources such as mutable references using type qualifiers, which enables richer programming patterns, *e.g.*, based on sharing captured mutable data, in languages that make pervasive use of higher order functions and type-level abstraction.

In prior work, key type soundness results for reachability types have been established [Bao et al. 2021; Wei et al. 2024] using the usual syntactic techniques of progress and preservation [Wright and Felleisen 1994] with respect to (*w.r.t.*) a small-step, substitution-based, operational semantics. These

Authors' Contact Information: Yuyan Bao, Augusta University, Augusta, USA, yubao@augusta.edu; Songlin Jia, Purdue University, West Lafayette, USA, jia137@purdue.edu; Guannan Wei, Tufts University, Medford and Somerville, USA, guannan.wei@tufts.edu; Oliver Bračevac, EPFL, Lausanne, Switzerland, oliver.bracevac@epfl.ch; Tiark Rompf, Purdue University, West Lafayette, USA, tiark@purdue.edu.

results assure us that reachability information is preserved through evaluation, serving, *e.g.*, as an informal justification that expressions with non-overlapping reachability can be safely evaluated in a different order or in parallel. Likewise, prior work [Bao et al. 2021; Wei et al. 2024] has proposed that reachability types could be used as the basis for effect systems to track (the absence of) store modifications in a fine-grained way, *e.g.*, to parallelize overlapping reads as long as there are no conflicting writes. However, no end-to-end formal proof of such properties exists so far, and in general, stronger metatheoretic properties beyond basic type safety have been left unexplored.

We address this gap by presenting an alternative semantic model of reachability types using the technique of logical relations [Ahmed et al. 2009; Benton et al. 2007; Plotkin 1973; Tait 1967; Timany et al. 2024], providing a framework in which to study key properties of interest such as semantic type soundness, termination, effect safety, and contextual equivalence.

Bottom-up Design of Logical Relations. We construct logical relations (LR) for a family of reachability type systems with increasing sets of features. Our starting point is the standard LR for the simply-typed λ-calculus with first-order references, which we augment with internal invariants to track the flow of store locations, without restricting expressiveness (Section 2). From these added invariants, we "rediscover" reachability types by propagating certain choices to qualifiers in the user-facing type system (Section 3). Then, we scale up the set of features, from first-order mutable references to higher-order mutable references (Section 3.5), from tracking all mentioned references uniformly to distinguishing reads and writes (Section 3.6), and from a unary LR to a binary LR (Section 4) to support equational reasoning (Section 5).

Semantic Type Soundness. The most immediate result from the LR model is semantic type soundness. As Timany et al. [2024] have argued, semantic type soundness is a stronger notion than syntactic type soundness, so our semantic type soundness result has significance beyond the existing syntactic results [Bao et al. 2021; Wei et al. 2024]. First, semantic type soundness does not require terms to be *syntactically* well-typed, just to *behave* in a well-typed way, *e.g.*, disregarding ill-typed branches that are never taken. Thus, it provides a foundation for studying potentially unsafe features and interactions with the outside world. Second, unlike prior results [Bao et al. 2021; Wei et al. 2024] that were established *w.r.t.* a small-step operational semantics based on substitution, we adopt a big-step operational semantics based on closures and environments. Thus, our results map more closely to realistic language implementations.

Termination. The LR model actually proves a stronger property than the absence of type errors, namely that evaluation of all well-typed terms *must terminate* with a well-typed value. This termination result is interesting in addition to soundness. First, the reachability type systems in Section 3.5 and beyond contain several non-trivial features such as nested mutable references and self references in function qualifiers, whose termination properties are non-obvious. Second, our LR model is defined by simple structural recursion on types, with a store typing invariant that precludes cycles in the store. This is in contrast to other recent work, which proposed a form of transfinite step indexing to prove termination for a feature-rich language with higher-order state [Spies et al. 2021]. In comparison, our model achieves a similar result but is entirely elementary and requires neither step-indexes, nor advanced set-theoretic concepts, nor classical reasoning. More broadly, the termination result is both reassuring, in the sense that reachability types can support flexible mutation in places where termination is required (*e.g.*, dependent type systems) but also points out a non-obvious limitation in prior work [Bao et al. 2021; Wei et al. 2024], namely the inability to construct cyclic heap structures such as doubly-linked lists. We leave it to future work to study this limitation and possible remedies further. Recently, promising steps in this direction were made by Deng et al. [2025].

Effect Safety. The built-in store invariants of the LR model guarantee that an expression can only observe or modify reachable store locations. Thus, reachability provides a natural upper bound on potential side effects. Specifically, the latent effect of a function is limited to the store locations reachable from the function and any locations reachable from the provided argument. In Section 3.6, we extend our LR model with an effect system that tracks *write effects* on sets of variables in addition to reachability. With this added precision of distinguishing pure observations from effectful write operations, we are able to prove an effect safety theorem, stating that all store changes are covered by write effects, *i.e.*, pure expressions never cause any observable effects. While similar extensions have been proposed in prior work [Bao et al. 2021; Bračevac et al. 2023a], no end-to-end proofs of effect safety have been provided so far. And in fact the desired properties are non-trivial to capture using progress and preservation, since effects are "performed" during reduction and not preserved in the types of values.

Program Equivalence. We extend our unary LR to a binary LR model in Section 4 to support equational reasoning, and we prove key equivalences in Section 5. This equational reasoning framework is significant as it provides a foundation for parallelization and for a variety of effect-based compiler optimizations in the style of Benton et al. [2007] or Birkedal et al. [2016]. First, we prove a reordering theorem for non-interfering expressions which justifies parallel execution. Second, we prove β-equivalence for functions applied to pure and non-interfering arguments. This result shows that the evaluation semantics used here is consistent with substitution (specifically, substitution with values), despite not using substitution internally. It also justifies function inlining as a compiler optimization, *e.g.*, in the context of Bračevac et al. [2023a]'s compiler IR based on reachability types for dependency analysis. We formulate equivalences for the calculus with and without effect qualifiers (Section 3.6) and show that write effects enable a more precise notion of non-interference than pure reachability.

Contributions. In summary, this paper makes the following contributions:

- We review the standard LR model for simply-typed λ-calculus with first-order mutable references, and discuss how adding certain store invariants leads to reachability types (Section 2).
- We present a unary logical relation that establishes semantic type soundness as well as termination for types with reachability qualifiers (Section 3).
- We extend the calculus to support higher-order mutable references and show that the system remains sound and terminating (Section 3.5).
- We extend the calculus with an effect system including an effect safety invariant, guaranteeing that pure expressions have no observable side effects (Section 3.6).
- We extend the model to a binary logical relation to support relational reasoning *w.r.t.* the observational equivalence of two programs (Section 4).
- We prove a reordering theorem for non-interfering expressions and β-equivalence for functions applied to pure and non-interfering arguments (Section 5).

The formal results in this paper have all been mechanized in Rocq. The developments are available online at https://github.com/tiarkrompf/reachability, together with an extended version of this paper [Bao et al. 2025].

2 Motivation

2.1 Brief Review of Reachability Types

Reachability types [Bao et al. 2021; Wei et al. 2024] are a recent proposal to bring Rust-style reasoning about memory properties to higher-level languages, specifically reasoning about sharing and the absence of sharing: separation. Reachability types are based on four key ideas:

(1) Tracking reachable variables in type qualifiers: Types are of the form T^p, where p is a *reachability qualifier*, a set of variables that may additionally include the *freshness marker* ♦.

```
val x = new Ref(0)       // : Ref[Int]^x in context [ x: (Ref Int)^♦ ]
val y = x                // : Ref[Int]^y in context [ y: (Ref Int)^x, x: (Ref Int)^♦ ]
```

The value of expression `new Ref(0)` is *fresh*: it must be tracked, but is not bound to a variable. The typing context $x : (\mathsf{Ref}\ \mathsf{Int})^{\blacklozenge}$ means x reaches a fresh value. The type system keeps reachability sets minimal in type assignment, *i.e.*, in the example, we assign the one-step reachability set y to the type of variable y. The complete reachability set `y,x` [1] can be retrieved by computing transitive closures *w.r.t.* the typing context [Wei et al. 2024].

Functions can reach all tracked values they close over. Thus, the reachability qualifier of a function includes the set of its free variables, consistent with the interpretation as a closure record:

```
def incr() = { x += 1 }  // : (() => Int)^x in context [ y: (Ref Int)^x, x: (Ref Int)^♦ ]
```

(2) Contextual freshness for function arguments: Reachability types can implement Rust-style borrowing that grants unique access to a store location while temporarily disabling other access paths. The following code defines a combinator `borrow`: [2]

```
// : ((z: (Ref Int)^♦) => ((Int => Int)^♦ => Unit)^z)^∅
def borrow(z: (Ref Int)^♦)(f: (Int => Int)^♦) = { z := f(!z) }
```

A fresh argument type means that the locations reachable from the argument and the function must be mutually disjoint, so that the function cannot *observe* any potential overlap with variables in the environment. This extends naturally to curried arguments such as `z`, `f` above:

```
val d = new Ref(0)
borrow(x)(y => y + !d)     // ok: argument type (Int => Int)^d is fresh relative to borrow(x)
borrow(x)(y => incr())     // type error: argument (Int => Int)^incr overlaps with borrow(x) via x
```

The first function call typechecks as the arguments are separate and fresh relative to the partially applied function `borrow(x)`, but the second does not, as both the passed closure and the partially applied function reach `x`.

(3) Self references to approximate reachable data for escaping values: The type system uses *self-references* in function types to track reachable values that escape from their defining scope. Our self references are similar to `this` pointers in OO languages, as formalized, *e.g.*, in the DOT family of type systems [Amin et al. 2016; Rompf and Amin 2016].

```
def cell(init: Int) = {  // : Int => (μf.() => (Ref Int)^f)^♦
  val c = new Ref(init)   // : (Ref Int)^c
  () => c                 // : (() => (Ref Int)^c)^c
}                        // : (μf.() => (Ref Int)^f)^♦ introduce self-ref to avoid local variable c
val z = cell(0)          // : (() => (Ref Int)^z)^z  instantiate self-ref f with bound name z
```

The above function `cell` returns a closure that captures and leaks the internal reference `c`. Once the closure escapes its defining scope, the variable `c` is no longer in scope, and thus cannot be mentioned in its type, as doing so would make the type meaningless to clients. [3] Thus, we use self-references in its type instead. The escaping closure is a fresh value before being bound to `z`. Its return type uses a self-reference introduced by the μ-notation (similar to DOT) to express that the returned value overlaps with the closure itself. Like `this` pointers in OO languages, self-references

[1]For readability, we often drop the set notation for qualifiers and write them down as comma-separated lists of atoms.
[2]Reachability types is a dependent type system, allowing a function's result type to mention its argument.
[3]The return type can only mention what is in scope, and needs to avoid any variable names going out of scope. This is called the "avoidance problem" in the context of ML modules and other languages with restricted forms of dependent types (*e.g.*, DOT [Amin et al. 2016; Rompf and Amin 2016]), where substitution is not always permitted.

Syntax $\boxed{\lambda_B}$

$$
\begin{array}{lcll}
S, T, U, V & ::= & Bool \mid T \to U \mid \mathsf{Ref}\ T & \text{Types} \\
c & := & \mathsf{true} \mid \mathsf{false} & \text{Constants} \\
t & ::= & c \mid x \mid \lambda x.t \mid t_1\ t_2 & \text{Terms} \\
 & \mid & \mathsf{ref}\ t \mid !\ t \mid t_1 \coloneqq t_2 \mid t_1; t_2 &
\end{array}
\qquad
\begin{array}{lcll}
v & ::= & c \mid \ell \mid \langle H,\ \lambda x.t \rangle & \text{Values} \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : T & \text{Typing Environment} \\
H & ::= & \varnothing \mid H; (x, v) & \text{Value Environment} \\
\sigma & := & \varnothing \mid \sigma; (\ell, v) & \text{Store}
\end{array}
$$

Typing Rules $\boxed{\Gamma \vdash t : T}$

$$
\frac{}{\Gamma \vdash c : Bool}\ (\text{T-CST})
\qquad
\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\text{T-VAR})
\qquad
\frac{\Gamma \vdash t : Bool}{\Gamma \vdash \mathsf{ref}\ t : \mathsf{Ref}\ Bool}\ (\text{T-REF})
\qquad
\frac{\Gamma \vdash t : \mathsf{Ref}\ Bool}{\Gamma \vdash !t : Bool}\ (\text{T-!})
$$

$$
\frac{\Gamma \vdash t_1 : \mathsf{Ref}\ Bool \quad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1 \coloneqq t_2 : Bool}\ (\text{T-:=})
\qquad
\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : Bool}{\Gamma \vdash t_1; t_2 : Bool}\ (\text{T-SEQ})
\qquad
\frac{\Gamma,\ x : T \vdash t : U}{\Gamma \vdash \lambda x.t : T \to U}\ (\text{T-ABS})
\qquad
\frac{\Gamma \vdash t_1 : T \to U \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : U}\ (\text{T-APP})
$$

Fig. 1. The syntax and static semantics of λ_B. We use type *Bool* as a representative type for primitive types, *e.g.*, TUnit and TInt. Thus, assignments (rule T-:=) return type *Bool*.

provide a way to tie hidden internal data to an externally visible identity, allowing us to track it throughout the interaction with the caller.

(4) Tracking effects: Reachability qualifiers provide a capability to reason about effects. Specifically, a function's qualifier is as an upper bound on what the function can observe or modify from the context. Effects can be tracked even more precisely if the type system is extended with a reachability-sensitive effect system [Bao et al. 2021; Bračevac et al. 2023a,b]. This enables, *e.g.*, distinguishing reads and writes on *the same* store reference. In this paper, we focus on tracking store modifications as *write effects* (Section 3.6). This allows us to distinguish pure from effectful code. This distinction also provides additional precision for program equivalence (Section 5). Consider the following example, where variables x and y are aliased.

```
val x = new Ref(5); val y = x; val z = new Ref(1)
y := y + !z           // :  @wr(y) in context [z: Ref[Int]♦, y: Ref[Int]×, x: Ref[Int]♦]
```

Incrementing the content of y by the value referenced by z induces a write effect (@wr) on variable y, and transitively on x too, but not on z. Thus, such an effect system can be used to show that values (*e.g.*, the referent of z) are preserved over a potential effectful computation [Bračevac et al. 2023b].

Towards a Semantic Model of Reachability Types. Having seen key examples of reachability types, we will now recap the structure of semantic type soundness via the standard λ-calculus LR definitions, and then proceed to "rediscover" reachability types from semantic store properties.

2.2 Semantic Soundness of λ_B

After introducing the standard LR model for simply-typed λ-calculus (STLC), we will add high-level semantic store invariants in Section 2.3 that track the flow of store-allocated values without restricting the set of typeable terms. Setting the scene for Section 3, we show how these semantic invariants naturally lead to the idea of exposing the corresponding information through user-facing type qualifiers, so that the flow of store-allocated values can be tracked and controlled by the programmer. We thus rediscover reachability types from first principles, and justify them in a semantic rather than syntactic way.

The λ_B Language. Fig. 1 shows the syntax and typing rules for λ_B, STLC with first-order store-allocated mutable references (restricted to hold Boolean values). The definitions are mostly standard, and can be found, e.g., in the popular TAPL textbook [Pierce 2004]. We use a big-step semantics with a value environment H and a store σ, where H is a partial function that maps variables to values, and σ is a partial function that maps locations to values. We write $t,\ H,\ \sigma \Downarrow v,\ \sigma'$ to denote that term t is evaluated to value v, resulting in a store transition from σ to σ'. The value environment

$\boxed{\lambda_B}$

Interpretation of Value Reachability

$$L(c) = \varnothing \qquad L(\ell) = \{\ell\} \qquad L(\langle H,\ \lambda x.t\rangle) = L(\mathrm{fv}(\lambda x.t))_H \qquad L(q)_H = \textstyle\bigcup_{x \in q} L(H(x))$$

Semantic Interpretation of Types, Terms, and Typing Contexts

$$\begin{aligned}
\Sigma &::= \varnothing \mid \Sigma, \ell : \mathbb{V} \\
\Sigma \sqsubseteq \Sigma' &\overset{\text{def}}{=} \forall\, \ell.\, \ell \in \mathrm{dom}(\Sigma) \Rightarrow (\ell \in \mathrm{dom}(\Sigma') \wedge \Sigma(\ell) = \Sigma'(\ell)) \\
\Sigma \sqsubseteq_{\mathbb{L}} \Sigma' &\overset{\text{def}}{=} \mathbb{L} \subseteq \mathrm{dom}(\Sigma) \wedge \mathbb{L} \subseteq \mathrm{dom}(\Sigma') \ \wedge\ (\forall\, \ell \in \mathbb{L}.\, \Sigma(\ell) = \Sigma'(\ell')) \\
\sigma : \Sigma &\overset{\text{def}}{=} \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \ \wedge\ (\forall \ell \in \mathrm{dom}(\sigma).\, \sigma(\ell) \in \Sigma(\ell)) \\
\sigma \rightarrow_L \sigma' &\overset{\text{def}}{=} \forall \ell \in \mathrm{dom}(\sigma).\, \ell \notin L \Rightarrow \sigma(\ell) = \sigma'(\ell)
\end{aligned}$$

$$\begin{aligned}
V[\![\mathrm{Bool}]\!] &= \{(H,\ \Sigma,\ c)\} \\
V[\![\mathrm{Ref\ Bool}]\!] &= \{(H,\ \Sigma,\ \ell) \mid \ell \in \mathrm{dom}(\Sigma) \ \wedge\ (\forall\, \Sigma'.\ (v \in \Sigma(\ell) \iff (H, \Sigma', v) \in V[\![\mathrm{Bool}]\!]))\}
\end{aligned}$$

$$\begin{aligned}
V[\![T \to U]\!] = \{&(H,\ \Sigma,\ \langle H',\ \lambda x.t\rangle) \mid \forall\, v,\ \sigma',\ \Sigma'.\ \textcircled{1}\, \Sigma \sqsubseteq_{L(\langle H',\ \lambda x.t\rangle)} \Sigma' \ \wedge\ \sigma' : \Sigma' \ \wedge \\
&\textcircled{2}\, L(v) \subseteq L(\langle H,\ \lambda x.t\rangle) \cup \overline{L(\langle H,\ \lambda x.t\rangle)} \ \wedge\ (H,\ \Sigma',\ v) \in V[\![T]\!] \Rightarrow \\
&\quad \exists\, \sigma'', \Sigma'', v'.\, t,\ H'; (x; v),\ \sigma' \Downarrow v',\ \sigma'' \ \wedge\ \sigma'' : \Sigma'' \ \wedge\ \Sigma' \sqsubseteq \Sigma'' \ \wedge\ (H,\ \Sigma'',\ v') \in V[\![U]\!] \ \wedge \\
&\quad \textcircled{3}\, L(v') \subseteq L(\langle H,\ \lambda x.t\rangle) \cup L(v) \cup \overline{\mathrm{dom}(\Sigma')} \ \wedge\ \textcircled{4}\, \sigma' \rightarrow_{(L(\langle H,\ \lambda x.t\rangle) \cup L(v))} \sigma'' \}
\end{aligned}$$

$$\begin{aligned}
E[\![T]\!] = \{&(H,\ \Sigma,\ t) \mid \forall\, \sigma.\, \sigma : \Sigma \ \wedge\ \exists\, \sigma', \Sigma', v'.\, t,\ H,\ \sigma \Downarrow v',\ \sigma' \ \wedge\ \sigma' : \Sigma' \ \wedge\ \Sigma \sqsubseteq \Sigma' \ \wedge \\
&(H,\ \Sigma',\ v') \in V[\![T]\!] \ \wedge\ L(v') \subseteq L(\mathrm{fv}(t))_H \cup \overline{\mathrm{dom}(\Sigma)} \ \wedge\ \sigma' \rightarrow_{L(\mathrm{fv}(t))_H} \sigma'' \}
\end{aligned}$$

$$\begin{aligned}
G[\![\Gamma^{\varphi}]\!] = \{&(H, \Sigma) \mid \mathrm{dom}(H) = \mathrm{dom}(\Gamma) \ \wedge\ \varphi \subseteq \mathrm{dom}(\Gamma) \ \wedge\ (\forall\, x, T.\, \Gamma(x) = T \ \wedge\ x \in \varphi \Rightarrow \\
&(H,\ \Sigma,\ H(x)) \in V[\![T]\!])\}
\end{aligned}$$

Semantic Typing Judgment $\qquad \Gamma \models t : T \ \overset{\text{def}}{=}\ \forall\, (H, \Sigma) \in G[\![\Gamma^{\mathrm{fv}(t)}]\!].\, (H,\ \Sigma,\ t) \in E[\![T]\!]$

Fig. 2. The value interpretation of types, terms, and typing context interpretation for λ_B. Formula $\textcircled{2}$ is a tautology here, but we will see that our reachability qualifiers can specify how argument values nontrivially interact with function values in Section 3. The highlighted formulas demonstrate that the knowledge about locations reachable from given values allows the proof to track properties of an effectful operation, *i.e.*, properties of locations unreachable during an operation will be preserved.

is immutable, as is standard. Evaluating each sub-term in a sequence $t_1; t_2$ term yields a Boolean value, and the overall result conjuncts the sub-terms' resulting values. [4] Other definitions of the semantics are standard, and can be found in the extended version of this paper [Bao et al. 2025].

Following [Amin and Rompf 2017; Ernst et al. 2006; Siek 2013; Wang and Rompf 2017], in the proof of semantic type soundness, we extend the big-step semantics $\Downarrow$ to a total evaluation function by adding a numeric fuel value and explicit timeout and error results. Note, however, that while the operational semantics is step-indexed in this way, the logical relations we define are not, and could be defined just as well with a partial (big-step or small-step) evaluation semantics.

Modeling Stores. Considering the restriction to first-order references here, the store layouts are always "flat", *i.e.*, free of cycles. To specify well-typed stores, we use a store typing Σ that maps from locations to semantic types. A semantic type is a set of values ($\mathbb{V}$). Observing the semantics, a store always grows monotonically during the course of evaluation, which is modeled by the store typing extension relation, written $\Sigma \sqsubseteq \Sigma'$. Store typing extension satisfies reflexivity and transitivity. Given a store σ, we write $\sigma : \Sigma$ to mean that σ is well-formed *w.r.t.* the store typing Σ. Fig. 2 (top, ignoring the highlighted formulas) summarizes the definitions of these notations.

[4] To verify that reordering two terms (rules in Section 5.1) yields equivalent results, we have to choose a commutative operation to combine the results. The most natural would be to combine the two results into a pair, but this would require either including pairs in the base language, or using a λ-encoding. Instead, we chose to allow each sub-term to yield a Boolean value, as it captures the essential property with less complexity than those alternatives approaches.

Value Interpretation of Types and Term Interpretation. Fig. 2 (middle) defines the value interpretation of types, as well as the term and typing context interpretations, ignoring the highlighted formulas for now. The value interpretation of type T, written as $V[[T]]$, is a set of triples of form $(H,\ \Sigma,\ v)$, where H is a value environment, Σ is a store typing, and v is a value.

The values of type Bool are true and false; the values of reference type Ref Bool are store locations that store a value of type Bool. The set of values of function type $T \to U$ *w.r.t.* store typing Σ is defined based on its computational behavior. It says that for all future stores $\sigma' : \Sigma'$, where $\Sigma \sqsubseteq \Sigma'$, for any value v of type T *w.r.t.* store typing Σ', if evaluating the function body in the extended value environment with store σ' results in a value of type U, a store σ'', and a store typing Σ'', such that $\sigma'' : \Sigma''$ and $\Sigma' \sqsubseteq \Sigma''$, then we conclude that the function values are valid at type $T \to U$ *w.r.t.* store typing Σ. Note that we inline the term interpretation to characterize the function body's computational behavior.

A term t has type T *w.r.t.* store typing Σ if for all $\sigma : \Sigma$, term t evaluates to a value of type T, a final store σ' and a store typing Σ', such that $\sigma' : \Sigma'$ and $\Sigma \sqsubseteq \Sigma'$.

The Compatibility Lemmas, Fundamental Theorem and Adequacy. The semantic typing judgment $\Gamma \models t : T$ is defined in Fig. 2 (bottom, ignoring the highlighted formula). It means that term t inhabits the term interpretation of type T under any value environment H and store typing Σ that satisfies the semantic interpretation of the typing context $G[[\Gamma]]$, as defined in Fig. 2 (bottom). The semantic typing rules have the same shape as the syntactic ones, but turning the symbol $\vdash$ (in Fig. 1) into the symbol $\models$. Each of the semantic typing rules is a compatibility lemma.

Theorem 2.1. *(Fundamental Theorem of Unary Logical Relations) Every syntactically well-typed term is semantically well-typed,* i.e., *if* $\Gamma \vdash t : T$, *then* $\Gamma \models t : T$.

The following theorem entails that a semantically typed term always terminates. The termination property is defined as part of the semantic typing judgment.

Theorem 2.2. *(Adequacy of Unary Logical Relations) Every closed semantically well-typed term t is safe: if* $\varnothing \models t : \mathsf{T}$, *then* $\exists\, v, \sigma.\, t,\ \varnothing,\ \varnothing \Downarrow v,\ \sigma$.

The fundamental theorem (Theorem 2.1) follows immediately from the compatibility lemmas, and Theorem 2.2 (Adequacy) and semantic type safety follow from the fundamental theorem (Theorem 2.1) and the definition of semantic term interpretation $\Gamma \models t : T$ and $E[[T]]$.

2.3 Rediscovering Reachability Types

The logical definition allows us to establish strong theoretical results such as semantic type soundness and termination but remains fairly limited in describing program behavior in other ways. Firstly, it does not allow us to prove the assertion true on the right, as the logical definition does not say anything about preserving store *values* across certain operations. Secondly, the assertion statement gets stuck if `!x == 1` is false, thus it is an unsafe feature [Timany et al. 2024], although the example uses it safely, which cannot be proven using the given logical definitions. [6] [7]

```
def m() {
  val x = new Ref(1)
  val y = new Ref(2)
  def f () = { y += 1 }
  f()
  assert (!x == 1)
}
```

Fig. 3. Safe use of unsafe feature encapsulated in function `m`.[5]

The highlighted formulas in Fig. 2 illustrate that information about reachable locations from given values can give us useful semantic store invariants. We first introduce function $L(v)$ (Fig. 2,

[5]The code that omits the assertion statement can be encoded as a sequence of λ-abstractions and applications in our systems.
[6]Those are not the general motivations behind reachability types, but rather to give an explanation for some requirements of an effective logical relations model. Please refer to prior works [Bao et al. 2021; Wei et al. 2024] for general motivation.
[7]See an informal justification of the safe use of the assertion statement in Fig. 3, using our binary logical relations in Section 4.3, but our goal is not primarily to verify safe use of general unsafe code like Jung et al. [2018a]'s work.

top) that computes reachable locations from value v. In Fig. 3, function `m` reaches what its free variables reach; *i.e.*, $\varnothing$.

With the knowledge of reachable locations, instead of using the standard definition of store typing extension, we introduce *relational store typing* (RST), written as $\Sigma \sqsubseteq_{\mathbb{L}} \Sigma'$, which parameterizes the definition of store typing extension with reachable locations $\mathbb{L}$. RST is symmetric, allowing Σ' to be defined from Σ, such that they agree on $\mathbb{L}$, leaving the remainder unspecified.[8] The standard definition of store typing extension $\Sigma \sqsubseteq \Sigma'$ can be defined as $\Sigma \sqsubseteq_{\mathrm{dom}(\Sigma)} \Sigma'$.

The Time Travelling Property. Now the reasoning can time travel between a past store typing and another (possible future) one back and forth, as long as they agree on reachable locations, *i.e.*, formula ①, which is characterized by the following lemma:

Lemma 2.3 (Time Travelling). *If* $(H, \Sigma, v) \in V[[T]]$, *and* $\Sigma \sqsubseteq_{L(v)} \Sigma'$, *then* $(H, \Sigma', v) \in V[[T]]$.

The lemma allows us to establish that the triple (H, Σ', v) is a valid element of $V[[T]]$ if we know (H, Σ, v) is valid as long as the two store typings agree on the locations reachable from value v.

In the interpretation of function types, formula ① states that types are only preserved *w.r.t.* reachable locations from function values, which gives us a degree of *local reasoning*, as Σ' does not assume anything from Σ that is not reachable from the function value. Under this assumption together with the argument value being semantically well-typed in Σ', the safe reduction of the function body implies that functions can only *access* locations reachable from their arguments and themselves. This characterizes a separation logic [O'Hearn et al. 2001; Reynolds 2002] style of reasoning, but without enforcing exclusiveness of ownership, uniqueness, or immutability.

In the example of Fig. 3, formula ① implies no store locations are needed in the pre-store (specified by Σ), characterizing proper encapsulation. In function `m`, function `f` reaches what `y` reaches, which is separate from what `x` reaches. Thus, we can prove the assertion true by formula ④.

In addition, with the reachability information, the value interpretation of function types can also know that an argument is either separate from the function, or has some overlap with the function (formula ②). This is a tautology here, but we will see that our reachability qualifiers can specify how argument values nontrivially interact with function values in Section 3. The return value may further reach fresh locations (formula ③). In the proof context, we know fresh locations are part of Σ'', so it is redundant to use $\Sigma'' - \Sigma'$ here. A similar refinement is adapted to the term interpretation. Now, for λ_B, the semantic typing judgment for open terms t can be localized to t's free variables, as formulated by the highlighted formula in Fig. 2 (bottom).

Having added these semantic store invariants to the logical relation, why don't we expose them to the programmer as features of the type system? This is precisely the idea of reachability types.

3 Semantic Type Soundness of $\lambda^{\blacklozenge}_{\mathrm{B}}$

In this section, we first present the reachability type system $\lambda^{\blacklozenge}_{\mathrm{B}}$ that tracks aliasing and expresses values reachable from a given expression's result, define its value interpretation of types and terms, and present the semantic typing rules and show semantic type soundness. Then, we present $\lambda^{\blacklozenge}$ (in Section 3.5) that extends $\lambda^{\blacklozenge}_{\mathrm{B}}$ to support higher-order mutable references. Finally, we present $\lambda^{\blacklozenge}_{\varepsilon}$ (in Section 3.6) that extends $\lambda^{\blacklozenge}$ with observable write effect qualifiers, demonstrating a stronger value preservation property than what is defined for $\lambda^{\blacklozenge}$.

3.1 The $\lambda^{\blacklozenge}_{\mathrm{B}}$ Language

Fig. 4 shows the parts of $\lambda^{\blacklozenge}_{\mathrm{B}}$'s syntax that differ from λ_B. A complete definition can be found in the extended version of this paper [Bao et al. 2025]. Terms follow λ_B, except that λ terms are in the form of $(\lambda x.t)^q$, carrying a reachability qualifier q that includes the function's captured variables.

[8]Starting from Section 3, we adopt the notation $\Sigma \equiv_L \Sigma'$ to reflect the symmetric nature of the RST.

Syntax $\boxed{\lambda^{\blacklozenge}_{\mathrm{B}}}$

r, s	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\} \uplus \{♤\})$	Function Domain/Codomain Qualifiers				
p, o	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var} \uplus \{\blacklozenge\})$	Reachability Qualifiers	φ, q	$\in$	$\mathcal{P}_{\mathsf{fin}}(\mathsf{Var})$	Observations
S, T, U, V	$::=$	$Bool \mid \mathsf{Ref}\ T^{p}$	Types	v	$::=$	$c \mid \ell \mid \langle H,\ (\lambda x.t)^{q} \rangle$	Values
	$\mid$	$(x : T^{s}) \to U^{r}$		Γ	$::=$	$\varnothing \mid \Gamma,\ x : T^{p}$	Typing Env

Fig. 4. The parts of $\lambda^{\blacklozenge}_{\mathrm{B}}$'s syntax that differ from λ_B. Differences from λ_B (in Fig. 1) are highlighted.

$\lambda^{\blacklozenge}_{\mathrm{B}}$ adds reachability qualifiers to its type language and allows dependent function types, which are in the form of $(x : T^{s}) \to U^{r}$, where both argument and return types are qualified. The codomain U^{r} may depend on the argument x in its qualifier r. To avoid dealing with deep substitution, we prohibit references to x from inside U. This restriction is purely technical; some prior works [Wei et al. 2024] do allow deep references, but this choice does not appear to impact expressiveness in any significant way [Jia et al. 2024] as deep references can also be modeled indirectly via chains of argument and self-references.

Following prior works on reachability types [Bao et al. 2021; Wei et al. 2024], types in $\lambda^{\blacklozenge}_{\mathrm{B}}$ are of the form T^{p}, where p is a *reachability qualifier*, a finite set of variables that may include the distinct freshness marker $\blacklozenge$, representing the values reachable from the given expression's result.

For simplicity, we introduce a self-reference marker ♤, rather than formalizing self-references λf as a binder of a λ term (as in prior works [Bao et al. 2021; Wei et al. 2024]). The self-reference marker ♤ may appear in the qualifiers of a function's argument and its result, often expressed by the symbols s and r as shown in Fig. 4. If present, it denotes that the corresponding value may reach what the function reaches. This simplified formalization does not change the use of self-references in prior works [Bao et al. 2021; Wei et al. 2024], but frees us from engaging in reasoning about recursion within the logical definitions, which could otherwise be done using well-established techniques such as step-indexing [Ahmed et al. 2009; Ahmed 2004; Appel and McAllester 2001].

Mutable reference types Ref T^{p} track the known aliases of the value held by the reference. We use *observation* φ, which is a finite set of variables, to specify which variables in the typing context Γ are observable, where the typing context assigns qualified typing assumptions to variables.

3.2 Store Typing and Semantic Typing Context

We refine the definition of store typings from Fig. 2 by adding reachable locations to each entry. Now, a store typing Σ maps from locations to pairs of semantic types and reachable locations, as shown in Fig. 5. A store σ is well-defined *w.r.t.* store typing Σ if each location ℓ in σ stores well-typed values (*w.r.t.* Σ), which reach the empty set of locations, *i.e.* $L(\sigma(\ell)) \subseteq \varnothing$. This condition reflects the restriction in $\lambda^{\blacklozenge}_{\mathrm{B}}$, where reference types are limited to the form of Ref $T^{\varnothing}$, *i.e.*, the referent of a reference does not reach any locations. This restriction is relaxed by $\lambda^{\blacklozenge}$ in Section 3.5.

Our logical definition is established on a well-defined logical program state, which is defined in the semantic typing context $G[\![\Gamma^{\varphi}]\!]$ shown in Fig. 5 (bottom). The logical program state specifies what can be observed by the observation φ, as anything outside the observation is irrelevant. In terms of encapsulation safety, the observation can also be considered as a boundary to another world. The semantic typing context defines two reachability invariants: Inv_o and Inv_S. Invariant Inv_o states that locations reachable from non-fresh values are bounded by those from its qualifier.

Invariant Inv_S states that reachability qualifier intersection distributes over locations *w.r.t.* qualifier saturation, which is written as $p{*}$. This invariant is critical for function applications, where one needs to ensure an argument value is permissible to invoke the function by inspecting the argument's qualifier against the prescribed separation/overlapping from a function's type. Following Wei et al. [2024]'s work, we assign minimal variable sets (called one-step reachability) and compute transitive closures when checking separation/overlapping. Fig. 5 (top) recaps their definitions of

Interpretation of Value Reachability $\boxed{\lambda^{\blacklozenge}_{\mathrm{B}}}$

$$L(c) = \varnothing \qquad L(\ell) = \{\ell\} \qquad L(\langle H,\ (\lambda x.t)^{q}\rangle) = [[q]]_H \qquad [[q]]_H = \textstyle\bigcup_{x\in q} L(H(x))$$

Variable Reachability and Qualifier Saturation $\boxed{\Gamma \vdash x \rightsquigarrow x}$ $\boxed{\Gamma \vdash q*}$

Reachability Relation $\Gamma \vdash x \rightsquigarrow y \Leftrightarrow x : T^{q,y} \in \Gamma$ Variable Saturation $\Gamma \vdash x* := \{\, y \mid x \rightsquigarrow^{*} y \,\}$

Qualifier Saturation $\Gamma \vdash q* := \bigcup_{x\in q} x*$

Unary Logical Relations

$$\begin{array}{rcl}
\Sigma & ::= & \varnothing \mid \Sigma, \ell : (\mathbb{V}, \mathbb{L}) \\
\Sigma \sqsubseteq_{\mathbb{L}} \Sigma' & \overset{\text{def}}{=} & \mathbb{L} \subseteq \mathrm{dom}(\Sigma) \wedge \mathbb{L} \subseteq \mathrm{dom}(\Sigma') \ \wedge\ (\forall\, \ell \in \mathbb{L}.\, \Sigma(\ell) = \Sigma'(\ell')) \\
\sigma : \Sigma & \overset{\text{def}}{=} & \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \ \wedge\ (\forall \ell \in \mathrm{dom}(\sigma).\, \exists\, \mathbb{V}.\Sigma(\ell) = (\mathbb{V}, \varnothing) \ \wedge\ \sigma(\ell) \in \mathbb{V} \ \wedge\ L(\sigma(\ell)) \subseteq \varnothing) \\
\mathtt{WF}(\Sigma, v) & \overset{\text{def}}{=} & L(v) \subseteq \mathrm{dom}(\Sigma) \\
\mathtt{WF}(p^{\Gamma}_{x}) & \overset{\text{def}}{=} & \tilde{p} \subseteq \mathrm{dom}(\Gamma_1) \qquad\qquad \text{where } \Gamma = \Gamma_1, x : T^{p}, \Gamma_2 \text{ for some } T \\
\sigma \longrightarrow_{L} \sigma' & = & \forall \ell \in \mathrm{dom}(\sigma).\, \ell \notin L \Rightarrow \sigma(\ell) = \sigma'(\ell)
\end{array}$$

$$\begin{array}{rcl}
V[[Bool]] & = & \{(H,\ \Sigma,\ c)\} \\
V[[\mathrm{Ref}\ T^{\varnothing}]] & = & \{(H,\ \Sigma,\ \ell) \mid \mathtt{WF}(\Sigma, \ell) \ \wedge\ (\exists\, \mathbb{V}, \Sigma(\ell) = (\mathbb{V}, \varnothing) \ \wedge\ (\forall\, \Sigma', v.\, L(v) \subseteq \varnothing \Rightarrow \\
 & & \quad (v \in \mathbb{V} \iff (H,\ \Sigma',\ v) \in V[[T]])))\}
\end{array}$$

$$\begin{array}{rcl}
V[[(x : T^{s}) \to U^{r}]] & = & \{(H,\ \Sigma,\ \langle H',\ (\lambda x.t)^{q}\rangle) \mid \mathtt{WF}(\Sigma, \langle H',\ (\lambda x.t)^{q}\rangle) \ \wedge\ (\forall\, v, \sigma', \Sigma'.\, \textcircled{1}\ \Sigma \equiv_{[[q]]_{H'}} \Sigma' \ \wedge \\
 & & \quad \sigma' : \Sigma' \ \wedge\ (H,\ \Sigma',\ v) \in V[[T]] \ \wedge\ \textcircled{2}\ L(v) \subseteq [[\tilde{s}]]_H \cup_{\spadesuit \in s} [[q]]_{H'} \cup_{\blacklozenge \in s} \overline{[[q]]_{H'}} \ \Rightarrow \\
 & & \qquad \exists\, \sigma'', \Sigma'', v'.\, H'; (x, v),\ \sigma',\ t \Downarrow v',\ \sigma'' \ \wedge\ \sigma'' : \Sigma'' \ \wedge\ \Sigma' \sqsubseteq \Sigma'' \ \wedge \\
 & & \qquad\quad (H,\ \Sigma'',\ v') \in V[[U]] \wedge \\
 & & \qquad\quad \textcircled{3}\ L(v') \subseteq ([[\tilde{r} \backslash x]]_H \cap [[q]]_{H'}) \cup_{x \in r} L(v) \cup_{\spadesuit \in r} [[q]]_{H'} \cup_{\blacklozenge \in r} \overline{\mathrm{dom}(\Sigma')}) \wedge \\
 & & \qquad\quad \textcircled{4}\ \sigma' \longrightarrow_{[[q]]_{H'} \cup L(v)} \sigma''\}
\end{array}$$

$$\begin{array}{rcl}
E[[T^{p}]]_{\varphi} & = & \{(H,\ \Sigma,\ t) \mid \forall\, \sigma.\, \sigma : \Sigma \ \wedge\ \exists\, \sigma', \Sigma', v'.\, H,\ \sigma,\ t \Downarrow v',\ \sigma' \ \wedge\ \sigma' : \Sigma' \ \wedge\ \Sigma \sqsubseteq \Sigma' \ \wedge \\
 & & \qquad (H,\ \Sigma',\ v') \in V[[T]] \ \wedge\ L(v') \subseteq [[\varphi \cap \tilde{p}]]_H \cup_{\blacklozenge \in p} \overline{\mathrm{dom}(\Sigma)} \ \wedge\ \sigma \longrightarrow_{[[\varphi]]_H} \sigma'\}
\end{array}$$

$$\begin{array}{rcl}
G[[\Gamma^{\varphi}]] & = & \{(H, \Sigma) \mid \mathrm{dom}(H) = \mathrm{dom}(\Gamma) \ \wedge\ \varphi \subseteq \mathrm{dom}(\Gamma) \ \wedge\ \mathrm{Inv}_{o}(\Gamma, H, \Sigma, \varphi) \ \wedge\ \mathrm{Inv}_{S}(H, \Sigma, \varphi)\} \\
\mathrm{Inv}_{o}(\Gamma, H, \Sigma, \varphi) & = & \forall\, x, T, p.\, \Gamma(x) = T^{p} \ \Rightarrow\ \mathtt{WF}(p^{\Gamma}_{x}) \ \wedge\ (x \in \varphi \ \Rightarrow\ (H,\ \Sigma,\ H(x)) \in V[[T]]) \wedge \\
 & & \qquad (\blacklozenge \notin p \ \Rightarrow L(H(x)) \subseteq [[\tilde{p}]]_H) \\
\mathrm{Inv}_{S}(H, \Sigma, \varphi) & = & \forall p, p'.\, \tilde{p} \subseteq \varphi \ \wedge\ \tilde{p'} \ \subseteq \varphi \ \Rightarrow [[\tilde{p}]]_H \cap [[\tilde{p'}]]_H \subseteq [[\tilde{p}* \cap \tilde{p'}*]]_H
\end{array}$$

Fig. 5. The value interpretation of types, terms and typing context interpretation for $\lambda^{\blacklozenge}_{\mathrm{B}}$. Differences from λ_B (in Fig. 2) are highlighted. Qualifiers s and r may include the self-references marker $\spadesuit$, and are used to specify function domain and codomain. The notation $\Sigma \equiv_{L} \Sigma'$ means the RST introduced in Section 2.

variable reachability and qualifier saturation, which compute all reachable variables transitively. In addition, the predicate $\mathtt{WF}(p^{\Gamma}_{x})$ for $x : T^{p} \in \Gamma^{\varphi}$ (defined in Fig. 5) prohibits cycles in qualifiers, as the typing context is extended with the binder (in a λ term), where a parameter's qualifier is in the same scope as the λ term. Thus, cycles in qualifiers are not syntactically well-formed.

3.3 The Logical Interpretation of Types and Terms

The definitions of logical interpretation of types and terms are shown in the middle of Fig. 5. They follow a similar logical structure as for λ_B in Fig. 2, but are enriched with the knowledge about reachability specified by the qualifiers in types (highlighted).

We use the notation $\cup_b$ to denote the conditional set union operator, *i.e.*, if the predicate b is true, the operator performs a union of the left- and right-hand operands; otherwise, the result defaults to the left-hand operand only. Let p be a reachability qualifier that may include the freshness marker $\blacklozenge$

and the self-reference marker $♢$. We write $\tilde{p}$ to denote dropping the freshness and the self-reference markers in p, resulting in the set of variables appearing in p, *i.e.*, $\tilde{p} := p \setminus (♦, ♢)$.

Interpretation of Reachability. Fig. 5 (top) refines function $L(v)$ that computes reachable locations from value v for $\lambda^{♦}_{\mathrm{B}}$. The definition follows that for λ_B (Fig. 2), except for function values. Here, reachable locations from a closure record $\langle H,\ (\lambda x.t)^q \rangle$ include those from its qualifier, *i.e.*, $[[q]]_H$, which computes reachable locations from variables in q. We often write $[[q]]_H$ for the reachable locations from a closure record when the context is clear.

Well-Defined Reachable Locations. To use the reachability information in logical definitions, we use the predicate $\mathtt{WF}(\Sigma, v)$ (shown in Fig. 5) to guarantee reachable locations from a value v are well-defined *w.r.t.* a store typing Σ, *i.e.*, $L(v) \subseteq \mathrm{dom}(\Sigma)$. For example, a location ℓ can only reach itself. Thus the predicate $\mathtt{WF}(\Sigma, \ell)$ checks $\ell \in \mathrm{dom}(\Sigma)$. For values of type *Bool*, their reachable locations are the empty set. So their well-defined predicates are trivially true, and thus are omitted from its value interpretation.

Reference Types. The value interpretation of reference type Ref $T^{\varnothing}$ are store locations that hold values of type T, and their contents reach the empty set of locations. Comparing with λ_B that only allows references to hold Boolean values, here, references can store any typeable values as long as they reach the empty set of locations.

Function Types. The logical interpretation of function type $(x : T^s) \to U^r$ is defined in Fig. 5, where s, T and U are closed under value environment H. Their closedness predicates are omitted in the figure, and in the logical definitions in the subsequent sections. The qualifier of the return value may include parameter x, as well as the freshness and self-reference markers. The qualifiers of a function and its argument specify how the argument value communicates with its function value (formula ②). Here, we discuss several examples in Wei et al. [2024]'s work against the formula. Consider the following reachability-polymorphic identity function:

```
def id(x: T^♦): T^{x} = x          // : ((x: T^♦) => T^{x})^∅
```

The type specifies that `id` cannot reach anything from its context ($q = \varnothing$ in ②), and accepts argument values that may reach separate locations from what `id` reaches ($s = ♦$ in ②). Thus, the `id` function accepts any values of type `T`.

A different parameter qualifier specifies permissible overlapping between functions and their argument values. Consider the following variants of `id` which capture some variable `z` in the context:

```
def id2(x: T^{♦}  ): T^{x} = { val u = z; x }     // : ((x: T^{♦}  ) => T^{x})^{z}
def id3(x: T^{♦,z}): T^{x} = { val u = z; x }     // : ((x: T^{♦,z}) => T^{x})^{z}
```

The qualifiers on the function types and their parameters specify the reachability assumption that the implementation can *observe* about its context ($q = \{z\}$ in ②), and about any given argument value, respectively. Function `id2` accepts arguments reaching anything that does not (directly or transitively) reach `z` ($s = ♦$ in ②), and `id3` permits `z` in the parameter's qualifier, *i.e.*, allowing any argument ($s = \{♦, z\}$ in ②).

The reachability qualifiers of return values express how they communicate with arguments and function values (formula ③). Consider the following function that mutates a captured reference cell and returns the argument (here, $q = \{c1\}$, $s = \{c1, ♦\}$). We annotate that the argument `x` is potentially aliased with the captured argument `c1` but apply the function with argument `c2`. Our type system would not propagate such imprecision by tracking one-step reachability. The return type only tracks the argument `x` ($r = \{x\}$ in ③), meaning the return value reaches what the argument reaches. When applying different arguments to the function, precise reachability is retained:

```
... // c1: T^{c1}, c2: T^{c2}
def incr(x: T^{c1,♦}): T^{x} = { c1 := !c1 + 1; x } // : ((x: T^{c1,♦}) => T^{x})^{c1}
incr(c1)                                           // : T^{c1}
```

Semantic Typing $\boxed{\Gamma^{\varphi} \models t : T^{p}}$

$$\frac{}{\Gamma^{\varphi} \models c : \mathit{Bool}^{\varnothing}}\ (\text{T-CST}) \qquad \frac{\Gamma(x) = T^{p} \qquad x \in \varphi}{\Gamma^{\varphi} \models x : T^{x}}\ (\text{T-VAR}) \qquad \frac{\Gamma^{\varphi} \models t_1 : (\mathsf{Ref}\ T^{\varnothing})^{p} \quad \Gamma^{\varphi} \models t_2 : T^{\varnothing}}{\Gamma^{\varphi} \models t_1 := t_2 : \mathit{Bool}^{\varnothing}}\ (\text{T-:=})$$

$$\frac{(\Gamma,\ x : T^{\varphi \cap (s[q/\spadesuit]) \cup_{\blacklozenge \in s} \blacklozenge})^{q',x} \models t : U^{r[q'/\spadesuit]} \qquad q' = \varphi \cap q \qquad \tilde{s} \subseteq q'}{\Gamma^{\varphi} \models (\lambda x.t)^{q'} : ((x : T^{s}) \to U^{r})^{q}}\ (\text{T-ABS})$$

$$\frac{\Gamma^{\varphi_1} \models t_1 : \mathit{Bool}^{q_1} \quad \Gamma^{\varphi_2} \models t_2 : \mathit{Bool}^{q_2} \quad \varphi_1 \subseteq \varphi \qquad \varphi_2 \subseteq \varphi}{\Gamma^{\varphi} \models t_1; t_2 : \mathit{Bool}^{\varnothing}}\ (\text{T-SEQ})$$

$$\frac{\Gamma^{\varphi} \models t_1 : ((x : T^{s}) \to U^{r})^{p} \quad \blacklozenge \notin s, o \qquad \Gamma^{\varphi} \models t_2 : T^{o} \qquad \tilde{o} \subseteq \tilde{s} \subseteq \varphi \qquad \tilde{r} \subseteq \varphi, x \qquad x \notin \mathrm{fv}(U)}{\Gamma^{\varphi} \models t_1\ t_2 : U^{r[o/x, p/\spadesuit]}}\ (\text{T-APP})$$

$$\frac{\Gamma^{\varphi} \models t : T^{p_1} \quad \tilde{p_1} \subseteq \tilde{p_2} \subseteq \mathrm{dom}(\Gamma) \quad p_3 = p_2 \cup_{\blacklozenge \in p_1 \vee \blacklozenge \in p_2} \blacklozenge}{\Gamma^{\varphi} \models t : T^{p_3}}\ (\text{T-SUB})$$

$$\frac{\Gamma^{\varphi} \models t_1 : ((x : T^{s}) \to U^{r})^{p} \quad \blacklozenge \in s \quad x \notin \mathrm{fv}(U) \qquad \Gamma^{\varphi} \models t_2 : T^{o} \qquad \spadesuit \notin s \Rightarrow \tilde{p}* \cap \tilde{o}* \subseteq \tilde{s} \quad \tilde{s} \subseteq \varphi \quad \tilde{r} \subseteq \varphi, x}{\Gamma^{\varphi} \models t_1\ t_2 : U^{r[o/x, p/\spadesuit]}}\ (\text{T-APP}\blacklozenge)$$

$$\frac{\Gamma^{\varphi} \models t : T^{\varnothing}}{\Gamma^{\varphi} \models \mathsf{ref}\ t : (\mathsf{Ref}\ T^{\varnothing})^{\blacklozenge}}\ (\text{T-REF}) \qquad \frac{\Gamma^{\varphi} \models t : (\mathsf{Ref}\ T^{\varnothing})^{p}}{\Gamma^{\varphi} \models !t : T^{\varnothing}}\ (\text{T-!}) \qquad \frac{\Gamma^{\varphi} \models t : U^{p} \quad x \in \tilde{p} \quad \Gamma(x) = T^{q} \quad q \subseteq \varphi}{\Gamma^{\varphi} \models t : U^{p[q/x]}}\ (\text{T-SUB-VAR})$$

Qualifier Substitution $\boxed{r[p/x]}$ $\boxed{r[q/\spadesuit]}$

$$\begin{array}{llll} r[p/x] = r \backslash \{x\} \cup p & x \in r & r[q/\spadesuit] = r \backslash \{\spadesuit\} \cup q & \spadesuit \in r \\ r[p/x] = r & x \notin r & r[q/\spadesuit] = r & \spadesuit \notin r \end{array}$$

Qualifier Shorthands

$$p, q := p \cup q \qquad x := \{x\} \qquad \blacklozenge := \{\blacklozenge\} \qquad \spadesuit := \{\spadesuit\} \qquad {}^{\blacklozenge\spadesuit}p := \{\blacklozenge\} \cup \{\spadesuit\} \cup p$$
$$p \cup_{b} q := b = \text{true}\ ?\ p, q : p \qquad \tilde{p} := p \backslash (\blacklozenge, \spadesuit)$$

Fig. 6. Semantic typing rules for $\lambda^{\blacklozenge}_{\mathrm{B}}$. Qualifiers s and r may include the self-references marker $\spadesuit$, and are used to specify function domain and codomain.

```
incr(c2)                                    // : T^{c2} ⟵ precision retained
```

The logical definition precisely captures our crucial design that has the freshness marker $\blacklozenge$ in qualifiers to explicitly communicate (non-)freshness, which is preserved by dependent application and substitution [Wei et al. 2024] (see rule T-APP$\blacklozenge$ in Fig. 6).

If the argument's qualifier includes both the self-reference and the freshness markers, any argument values of the proper pretype are allowed. If the self-reference marker appears in the return value's qualifier, as in the `cell` example in Section 2.1, it means the return value can reach whatever the returning closure reaches.

The Term Interpretation. The term interpretation also adds the interpretation of reachability for its result values v. If the result value v is not fresh, then we check that its reachable locations are bounded by those from the reachability qualifier observed by the observation φ, *i.e.*, $\varphi \cap \tilde{p}$; otherwise, it may additionally include fresh locations $\overline{\mathrm{dom}(\Sigma)}$. The preserved store values are those not reachable from observation φ.

3.4 The Compatibility Lemmas, Fundamental Theorem and Adequacy

The semantic typing judgment is defined as $\Gamma^{\varphi} \models t : T^{p} \overset{\text{def}}{=} \forall (H, \Sigma) \in G[[\Gamma^{\varphi}]].\,(H,\ \Sigma,\ t) \in E[[T^{p}]]_{\varphi}$, meaning that term t inhabits the term interpretation of type T^{p} under any value environment H and store typing Σ that satisfies the semantic interpretation of typing context $G[[\Gamma^{\varphi}]]$.

Fig. 6 shows the semantic typing rules. For readability, we often drop the set notation for qualifiers and write them down as comma-separated lists of atoms. Each of the rules is a compatibility lemma, and has been proved in Rocq. The syntactic typing judgments have the form $\Gamma^{\varphi} \vdash t : T^{p}$. The syntactic typing rules have the same shape as the semantic one, but turning the symbol $\models$ (in Fig. 6) into the symbol $\vdash$.

We have proved the compatibility lemmas, fundamental theorem and adequacy for $\lambda^{\blacklozenge}_{\mathrm{B}}$. The fundamental theorem follows immediately from the compatibility lemmas, and adequacy, semantic type safety, and termination follow from the fundamental theorem and the definition of semantic term interpretation in Fig. 5.

To keep the exposition consistent with prior work [Wei et al. 2024], we extend $\lambda^{\blacklozenge}_{\mathrm{B}}$ with subtyping. See the extended version of this paper for the details [Bao et al. 2025].

Encapsulation. Now, we study encapsulation property demanded for rule β-EQUIV in Section 5. The typing judgment $\Gamma^{\varnothing} \vdash t : T^{\varnothing}$ expresses *encapsulated computations*, which are logically characterized by the following lemma:

Lemma 3.1 (Encapsulated Computations ($\lambda^{\blacklozenge}_{\mathrm{B}}$)). *If* $\Gamma^{\varnothing} \vdash t : T^{\varnothing}$, *then* $\forall\,(H, \Sigma) \in G[\![\Gamma^{\varnothing}]\!].\ \forall\,\Sigma'.\ \Sigma \equiv_{\varnothing} \Sigma' \Rightarrow (H, \Sigma', t) \in E[\![T^{\varnothing}]\!]_{\varnothing}$.

The lemma expresses that expression t is valid at type $T^{\varnothing}$ *w.r.t.* arbitrary store typing, including the empty one. Note that term t can be an implementation of an abstract data type, which may internally allocate new store locations. The empty qualifier (in $T^{\varnothing}$) dictates that t's evaluation result cannot be tracked, hence are encapsulated.

3.5 Extension with Higher-Order Mutable References ($\lambda^{\blacklozenge}$)

As presented so far, $\lambda^{\blacklozenge}_{\mathrm{B}}$ only supports *first-order* mutable references, *i.e.*, mutable references cannot hold values that contain references to other store locations. In this section, we introduce $\lambda^{\blacklozenge}$, which extends $\lambda^{\blacklozenge}_{\mathrm{B}}$ to support higher-order mutable references. We outline the major differences from $\lambda^{\blacklozenge}_{\mathrm{B}}$ here. The complete definitions can be found in the extended version of this paper [Bao et al. 2025].

The $\lambda^{\blacklozenge}$ Language. We relax the restriction on referent type $T^{\varnothing}$ for mutable references, *i.e.*, a referent's type can now carry a reachability qualifier T^{q}, indicating that the referent is a non-fresh value of type T, and may reach locations specified by q.

Following $\lambda^{\blacklozenge}_{\mathrm{B}}$, a store typing maps from locations to pairs of semantic types and reachable locations. Let ℓ be a location in store typing Σ, such that $\Sigma(\ell) = (\mathbb{V}, \mathbb{L})$, for some $\mathbb{V}$ and $\mathbb{L}$, where $\mathbb{V}$ is its semantic type, and $\mathbb{L}$ prescribes the set of locations reachable from the value stored in location ℓ. As location ℓ can indirectly reach some locations in a store, following prior work [Wei et al. 2024], we use location saturation, written as $\mathbb{L}^{*}_{\Sigma}$, to transitively compute reachable sets, and demand that the saturated reachable locations from the content is a subset of the saturated ones from $\mathbb{L}$. In addition, following the syntactic typing rules that disallow cyclic references [Wei et al. 2024], our logical definition requires that earlier allocated locations cannot reach those allocated later, which is defined in $\sigma : \Sigma$, *i.e.*, $\forall \ell' \in \mathbb{L}.\ \ell' < \ell$. Here, store locations are ordered by their allocation time.

The Logical Interpretation of Types and Terms. The definitions of the value interpretation of types, terms, and typing contexts follow $\lambda^{\blacklozenge}_{\mathrm{B}}$, but taking location saturation into consideration. For example, the predicate $\mathtt{WF}(\Sigma, v)$ is refined to ensure all the indirectly reachable locations are well-defined *w.r.t.* a store typing, *i.e.*, $\mathtt{WF}(\Sigma, v) = L(v)^{*}_{\Sigma} \subseteq \mathrm{dom}(\Sigma)$.

Similarly, in the definition of value interpretation of Boolean and function types, and term interpretation, the reachability properties defined by $L(v)$ and $[\![q]\!]_{H}$ for $\lambda^{\blacklozenge}_{\mathrm{B}}$ are now defined using location saturation, *i.e.*, $L(v)^{*}_{\Sigma}$ and $[\![q]\!]^{H*}_{\Sigma}$ for some store typing Σ. For example, formula ② in Fig. 5 is refined to $L(v)^{*}_{\Sigma'} \subseteq [\![\tilde{s}]\!]^{H'*}_{\Sigma'} \cup_{\diamondsuit \in s} [\![q]\!]^{H'*}_{\Sigma'} \cup_{\blacklozenge \in s} \overline{[\![q]\!]^{H'*}_{\Sigma'}}$.

Values of reference type Ref T^{q} are store locations ℓ, where their contents may change over time, but must always be of type T. The qualifier q allows the reasoning to time travel back and forth between two store typings as long as they agree on the reachable locations from the referent. Thus, the value interpretation of reference types in Fig. 5 is refined to $(\forall\,\Sigma', v.\ \Sigma \equiv_{\mathbb{L}^{*}_{\Sigma}} \Sigma' \wedge L(v)^{*}_{\Sigma'} \subseteq \mathbb{L}^{*}_{\Sigma'} \Rightarrow$

Semantic Typing

$$\frac{\Gamma^{\varphi} \models t : T^{q} \qquad q \subseteq \varphi}{\Gamma^{\varphi} \models \text{ref } t : (\text{Ref } T^{q})^{\blacklozenge q}} \ (\text{T-REF}) \qquad \frac{\Gamma^{\varphi} \models t : (\text{Ref } T^{q})^{p} \qquad q \subseteq \varphi}{\Gamma^{\varphi} \models !t : T^{q}} \ (\text{T-!}) \qquad \frac{\Gamma^{\varphi} \models t_1 : (\text{Ref } T^{p_1})^{p} \quad \Gamma^{\varphi} \models t_2 : T^{\tilde{p_1}} \qquad \tilde{p_1} \subseteq \varphi}{\Gamma^{\varphi} \models t_1 := t_2 : Bool^{\varnothing}} \ (\text{T-:=})$$

Fig. 7. Selected semantic typing rules for $\lambda^{\blacklozenge}$. Differences from $\lambda^{\blacklozenge}_{\mathrm{B}}$ are highlighted.

$(v \in \mathbb{V} \iff (H, \Sigma', v) \in V[[T]]))$, where $\mathbb{L}^{*}_{\Sigma}$ is the logical interpretation of q, *i.e.*, $\mathbb{L} = [[q]]_{H}$. The definition can be found in the extended version of this paper [Bao et al. 2025].

The Time Travelling Property. The time travelling lemma (Lemma 2.3 in Section 2) is refined with saturated locations as follows:

Lemma 3.2 (Time Travelling ($\lambda^{\blacklozenge}$)). *If* $(H, \Sigma, v) \in V[[T]]$, *and* $\Sigma \equiv_{L(v)^{*}_{\Sigma}} \Sigma'$, *then* $(H, \Sigma', v) \in V[[T]]$. The lemma allows the validity of the value v at type T to be preserved from store typing Σ to another Σ', as long as they agree on the saturated locations reachable from value v.

The Compatibility Lemmas, Fundamental Theorem and Adequacy. The definition of semantic typing judgment follows what was defined for $\lambda^{\blacklozenge}_{\mathrm{B}}$ in Fig. 5. Fig. 7 shows the selected semantic typing rules. Following prior work [Wei et al. 2024], the type system supports a restricted form of nested references: only non-fresh values can be stored/read from/written into in a reference (T-REF, T-! and T-:=). As referent qualifiers are invariant, store typing is monotonic. Each of the semantic typing rules is a compatibility lemma, and has been proven in Rocq.

We have proved the compatibility lemmas, fundamental theorem and adequacy for $\lambda^{\blacklozenge}$. The fundamental theorem follows immediately from the compatibility lemmas, and adequacy, semantic type safety, and termination follow from the fundamental theorem and the definition of semantic term interpretation.

Encapsulation. The encapsulated computation lemma for $\lambda^{\blacklozenge}_{\mathrm{B}}$ (Lemma 3.1) remains valid for $\lambda^{\blacklozenge}$. Its formalization is similar, and thus is omitted.

Discussion 1: Modeling Key Features in Prior Work. Our logical relations for $\lambda^{\blacklozenge}$ models a major subset of Wei et al. [2024]'s $\lambda^{\blacklozenge}$ system. We summarize the differences as follows:

(1) Self-References: As mentioned previously, we use a self-reference marker ♤, instead of formalizing self-references λf as a binder of a λ term as in Wei et al. [2024]'s work. However, this simplified formalization retains their key feature, *i.e.*, the lightweight form of qualifier polymorphism which is achieved by rule T-APP♦ in both works.

(2) Deep Substitution: As mentioned previously, to simplify the formalization, our T-APP rule does not consider deep substitution as theirs does. This restriction is purely technical, and does not appear to impact expressiveness in any significant way [Jia et al. 2024] as deep references can also be modeled indirectly via chains of argument and self-references.

(3) Reference Rules: In addition, our references rules require that a reference's qualifier be bounded by observation φ. This property is not explicitly specified in their system, but can be inferred from their tying rules. Thus, our reference rules are semantically equivalent to theirs.

Discussion 2: Termination. Our semantic is parameterized over a fuel value that bounds the steps that the execution is allowed; if it runs out of fuel, timeout will be returned. In this setting, we prove termination by providing enough fuel for the execution. In addition, proving termination in the presence of higher-order mutable references requires some constraints: the qualifiers attached to a store location prohibit storing any values that may reach the location itself, thus excluding cycles. This is a limitation in prior work [Wei et al. 2024] we discovered, leading to programs like Landin's knot [Landin 1964] not being typeable. See [Bao et al. 2025] for details.

Discussion 3: Strong Type Soundness. Prior works [Bao et al. 2021; Wei et al. 2024] proved syntactic type soundness using small-step semantics, where reachability properties are checked

$\boxed{\lambda^{\blacklozenge}_{\varepsilon}}$

Unary Logical Relations

$$V[[(x : T^{s}) \xrightarrow{\varepsilon} U^{r}]] = \{(H,\ \Sigma,\ \langle H',\ (\lambda x.t)^{q}\rangle) \mid \cdots \wedge \sigma' \longrightarrow_{[[\varepsilon \backslash x]]^{H*}_{\Sigma'} \cup_{x \in \varepsilon} v^{*}_{\Sigma'} \cup_{\diamondsuit \in \varepsilon} [[q]]^{H'*}_{\Sigma'}} \sigma''\}$$

$$E[[T^{p}\ \varepsilon]]_{\varphi} = \{(H,\ \Sigma,\ t) \mid \cdots \wedge \sigma \longrightarrow_{[[\varphi \cap \varepsilon]]^{H*}_{\Sigma}} \sigma'\}$$

Semantic Typing Rules

$$\frac{\Gamma^{\varphi} \models t_1 : \left(x : T^{s} \xrightarrow{\varepsilon_3} U^{r}\right)^{p}\ \varepsilon_1 \qquad \theta = [o/x, p/\diamondsuit] \qquad \Gamma^{\varphi} \models t_2 : T^{o}\ \varepsilon_2 \qquad \diamondsuit \notin s \Rightarrow p* \cap \tilde{o}* \subseteq \tilde{s} \\ \blacklozenge \in s \qquad \tilde{s} \subseteq \varphi \qquad x \notin \mathrm{fv}(U) \qquad \tilde{\varepsilon_3} \subseteq \varphi, x \qquad \tilde{r} \subseteq \varphi, x \qquad \theta' = [\tilde{o}/x, \tilde{p}/\diamondsuit]}{\Gamma^{\varphi} \models t_1\ t_2 : U^{r\theta}\ (\varepsilon_1 \rhd \varepsilon_2 \rhd \varepsilon_3)\theta'} \quad (\text{E-APP-}\blacklozenge)$$

$$\frac{\Gamma^{\varphi} \models t_1 : (\mathsf{Ref}\ T^{p_1})^{p}\ \varepsilon_1 \qquad \Gamma^{\varphi} \models t_2 : T^{\tilde{p_1}}\ \varepsilon_2 \qquad \tilde{p_1} \subseteq \varphi}{\Gamma^{\varphi} \models t_1 := t_2 : Bool^{\varnothing}\ \varepsilon_1 \rhd \varepsilon_2 \rhd \tilde{p}} \quad (\text{E-:=}) \qquad \frac{\Gamma^{\varphi} \models t : (\mathsf{Ref}\ T^{q})^{p}\ \varepsilon \qquad q \subseteq \varphi}{\Gamma^{\varphi} \models !t : T^{q}\ \varepsilon} \quad (\text{E-!})$$

Syntax $\varepsilon \in \mathcal{P}_{\mathrm{fin}}(\mathrm{Var} \uplus \{\diamondsuit\})$ **Flow-Insensitive Sequential Effects Composition** $\varepsilon_1 \rhd \varepsilon_2 := \varepsilon_1 \cup \varepsilon_2$

Fig. 8. Effect interpretation and selected typing rules for $\lambda^{\blacklozenge}_{\varepsilon}$. Differences from $\lambda^{\blacklozenge}$ are highlighted. Effect qualifiers (ε) are finite set of variables, and may include the self-reference marker $\diamondsuit$. The effect composition (*e.g.*, $\varepsilon_1 \rhd \varepsilon_2$) is defined as set union (*i.e.*, $\varepsilon_1 \cup \varepsilon_2$).

for intermediate values through their preservation theorems. In contrast, our work extends the big-step semantics $\Downarrow$ to a total evaluation function, making a distinction between timeout, errors, and normal values, leading to strong soundness results [Amin and Rompf 2017]. In particular, we define reachability properties as invariants in the value interpretation of types and terms in the logical framework. The fundamental theorem is proven by induction on the type derivation ($\Gamma^{\varphi} \vdash t : T^{q}$), where each case is the corresponding compatibility lemma (*i.e.*, semantic typing rule). Therefore, the reachability properties are also verified for each sub-expression.

3.6 Write Effect System Extension ($\lambda^{\blacklozenge}_{\varepsilon}$)

In the previously presented frameworks, any reachable location could be modified while evaluating an expression. This section presents $\lambda^{\blacklozenge}_{\varepsilon}$ that adds observable write effect qualifiers to track which locations are modified, as opposed to just read or, *e.g.*, passed as argument without being dereferenced. This gives us a stronger store value preservation property than previous frameworks.

Effect Types. The syntax of effect qualifiers are shown in Fig. 8 (bottom). Effect qualifiers (ε) are finite sets of variables, and may include the self-reference marker $\diamondsuit$, denoting the locations (in the pre-state) that may be modified during computation. Function types carry effect qualifiers ε for their latent effects.

Logical Interpretation of Effects. Fig. 8 shows the interpretation of function types and terms that includes effects, which are highlighted in the figure. Consider the following example:

```
... // c1: T^c1, c2: T^c2
def f (x: T^♦) = { c1 := !x + 1 }  // : ((x:T^♦) =>^c1 ())^{c1}
f(c2)                              // : () c1
```

The function `f` has its latent effect on the variable `c1`, meaning that its body can only modify the locations reachable from `c1`, as shown in the highlighted formula in Fig. 8. When the function is invoked, *i.e.*, `f(c2)`, the effect is just `c1` as well, according to rule E-APP-$\blacklozenge$. In contrast, without the effect extension, $\lambda^{\blacklozenge}$ approximates the write effect to what the argument and function reaches. As a result, `f(c2)` would induce effect on both `c1` and `c2`, leading to a weaker value preservation property, as illustrated by formula ④ and term interpretation in Fig. 5.

Other definitions follow $\lambda^{\blacklozenge}$, and are omitted in Fig. 8. See the extended version of this paper for the complete definitions [Bao et al. 2025].

Semantic Typing Judgment and Rules. The definition of the semantic typing judgment is defined as: $\Gamma^{\varphi} \models t : T^{p}\ \varepsilon \overset{\text{def}}{=} \forall\, (H, \Sigma) \in G[[\Gamma^{\varphi}]].\ (H,\ \Sigma,\ t) \in E[[T^{p}\ \varepsilon]]_{\varphi}$. It means that term t inhabits the term interpretation of type $T^{p}\ \varepsilon$ under any value environment H and store typing Σ that satisfies the semantic interpretation of the typing context $G[[\Gamma^{\varphi}]]$, which follows $\lambda^{\blacklozenge}$, and can be found in the extended version of this paper [Bao et al. 2025].

Fig. 8 shows the selected semantic typing rules for $\lambda^{\blacklozenge}_{\varepsilon}$, extending those of $\lambda^{\blacklozenge}$ with observable write effect qualifiers (ε), denoting the set of locations (in the pre-state) that may be modified during the execution. The effect rules are straightforward: the final effect of a compound term combines the effects of sub-terms with the intrinsic effect of this term. For example, the effects of assignments (rule E:=) consist of two parts: (1) the effects ε_1 and ε_2, which are those of sub-terms, and (2) the effect $\tilde{p}$, which denotes the reachable locations from variables in $\tilde{p}$ being modified. The final effects are sequential composition of those effects. In a flow-insensitive effect system like $\lambda^{\blacklozenge}_{\varepsilon}$, the composition is defined as set union . [9] See [Bao et al. 2025] for the complete rules.

Fundamental Theorem and Adequacy. The syntactic typing judgments have the form $\Gamma^{\varphi} \vdash t : T^{p}\ \varepsilon$, and the typing rules have the same shape as the semantic ones, but turning the symbol $\models$ into the symbol $\vdash$. We have proved the compatibility lemmas, fundamental theorem and adequacy for $\lambda^{\blacklozenge}_{\varepsilon}$. The fundamental theorem follows immediately from the compatibility lemmas, and adequacy, semantic type safety, and termination follow from the fundamental theorem and the definition of semantic term interpretation.

Encapsulation. The encapsulated computation lemma for $\lambda^{\blacklozenge}_{\varepsilon}$ demand that an encapsulated computation does not induce observable effects. See [Bao et al. 2025] for its formalization.

4 Contextual Equivalence via Binary Logical Relations

We now switch from modeling soundness, termination, and other properties of a *single* expression to properties concerning *pairs* of expressions, specifically notions of *observational equivalence*. Thus, we extend our unary logical relations for $\lambda^{\blacklozenge}$ and $\lambda^{\blacklozenge}_{\varepsilon}$ to binary logical relations, with the intention that semantically equivalent values and terms are exactly the ones that will be logically related. We formalize their judgments into one, as $\lambda^{\blacklozenge}_{\varepsilon}$ only adds conditions regarding effects. For example, the typing judgment is written as $\Gamma^{\varphi} \models T^{p}\ \lceil\varepsilon\rfloor$, where formula $\lceil\varepsilon\rfloor$ applies only to $\lambda^{\blacklozenge}_{\varepsilon}$.

4.1 Contextual Equivalence

Two programs are contextually equivalent if a well-typed program context cannot distinguish between them, *i.e.*, they have the same observable behavior. Here, we define contextual equivalence to verify equational rules (Section 5). A program t_1 is said to be *contextually equivalent* to another program t_2, written as $\Gamma^{\varphi} \models t_1 \approx_{\text{ctx}} t_2 : T^{p}\ \lceil\varepsilon\rfloor$, if for any program context C with a hole of type $T^{p}\ \lceil\varepsilon\rfloor$, if $C[t_1]$ has some (observable) behavior, then so does $C[t_2]$. The definition of context C is shown soon. Following prior works [Ahmed et al. 2009; Timany et al. 2024], we define a judgement for logical equivalence using binary logical relations, written as $\Gamma^{\varphi} \models t_1 \approx_{\log} t_2 : T^{p}\ \lceil\varepsilon\rfloor$.

Unlike reduction contexts, contexts C for reasoning about equivalence allow a “hole” to appear in any place. We write $C : (\Gamma^{\varphi}; T^{p}\ \lceil\varepsilon\rfloor) \Rrightarrow (\Gamma'^{\varphi'}; T'^{p'}\ \lceil\varepsilon'\rfloor)$ to mean that the context C is a program of type $T'^{p'}\ \lceil\varepsilon'\rfloor$ (closed under $\Gamma'^{\varphi'}$) with a hole that can be filled with any program of type $T^{p}\ \lceil\varepsilon\rfloor$ (closed under Γ^{φ}). The typing rules for well-typed contexts imply that if $\Gamma^{\varphi} \vdash t : T^{p}\ \lceil\varepsilon\rfloor$ and $C : (\Gamma^{\varphi}; T^{p}\ \lceil\varepsilon\rfloor) \Rrightarrow (\Gamma'^{\varphi'}; T'^{p'}\ \lceil\varepsilon'\rfloor)$ hold, then $\Gamma'^{\varphi'} \vdash C[t] : T'^{p'}\ \lceil\varepsilon'\rfloor$. Selected typing rules for well-typed contexts can be found in the extended version of this paper [Bao et al. 2025].

[9] One could also incorporate a flow-sensitive effect system, as suggested by Bao et al. [2021], to model more complex effects.

Relational Worlds, Well-Defined Relations and Store Typing $\boxed{\lambda^{\blacklozenge}/\lambda^{\blacklozenge}_{\varepsilon}}$

$$
\begin{aligned}
\Sigma &::= \varnothing \mid \Sigma, \ell : \mathbb{L} \\
\sigma : \Sigma &\overset{\text{def}}{=} \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \wedge (\forall \ell \in \mathrm{dom}(\Sigma). L(\sigma(\ell))^{*}_{\Sigma} \subseteq \Sigma(\ell)^{*}_{\Sigma} \wedge (\forall \ell' \in \Sigma(\ell).\, \ell' < \ell)) \\
\\
\mathrm{WR}(\mathrm{W}, \mathbb{L}, \mathbb{L}') &\overset{\text{def}}{=} (\mathbb{L}, \mathbb{L}') \subseteq \mathrm{dom}(\mathrm{W}) \wedge (\forall \ell_1, \ell_2,\ (\ell_1, \ell_2) \in f \Rightarrow (\ell_1 \in \mathbb{L} \iff \ell_2 \in \mathbb{L}')) \\
\mathrm{W} \equiv_{(\mathbb{L},\mathbb{L}')} W' &\overset{\text{def}}{=} \mathrm{WR}(\mathrm{W}, \mathbb{L}, \mathbb{L}') \wedge \mathrm{WR}(\mathrm{W}', \mathbb{L}, \mathbb{L}') \wedge \mathrm{W}_1 \equiv_{\mathbb{L}} \mathrm{W}'_1 \wedge \mathrm{W}_2 \equiv_{\mathbb{L}'} \mathrm{W}'_2 \wedge \\
&\quad\ (\forall\, \ell_1 \in \mathbb{L},\ \ell_2 \in \mathbb{L}'.\, \mathrm{W}(\ell_1, \ell_2) \Rightarrow \mathrm{W}_{\mathbb{V}}(\ell_1, \ell_2) = \mathrm{W}'_{\mathbb{V}}(\ell_1, \ell_2)) \wedge \\
&\quad\ (\forall\, \ell_1 \in \mathbb{L},\ \ell_2 \in \mathbb{L}'.\, (\ell_1, \ell_2) \in \mathrm{W} \leftrightarrow (\ell_1, \ell_2) \in \mathrm{W}') \\
(\sigma_1, \sigma_2) : \mathrm{W} &\overset{\text{def}}{=} \sigma_1 : \mathrm{W}_1 \wedge \sigma_2 : \mathrm{W}_2 \wedge (\forall \ell_1, \ell_2,\ \mathrm{W}(\ell_1, \ell_2) \Rightarrow (\sigma(\ell_1), \sigma(\ell_2)) \in \mathrm{W}_{\mathbb{V}}(\ell_1, \ell_2) \wedge \\
&\quad\ \mathrm{WR}(\mathrm{W}, \sigma_1(\ell_1)^{*}_{\mathrm{W}_1}, \sigma_2(\ell_2)^{*}_{\mathrm{W}_2}))
\end{aligned}
$$

Fig. 9. Definitions of relational worlds, well-defined relations and store typing for binary logical relations.

Two well-typed terms, t_1 and t_2, under typing context Γ^{φ}, are *contextually equivalent* if any occurrences of the first term in a closed term can be replaced by the second term without affecting the *observable results* of reducing the program, which is formally defined as follows:

Definition 4.1 (Contextual Equivalence). *Term* t_1 *is* contextually equivalent *to* t_2, *written* $\Gamma^{\varphi} \models t_1 \approx_{ctx} t_2 : T^{p}\ \lfloor\boldsymbol{\varepsilon}\rceil$, *if* $\Gamma^{\varphi} \vdash t_1 : T^{p}\ \lfloor\boldsymbol{\varepsilon}\rceil$, *and* $\Gamma^{\varphi} \vdash t_2 : T^{p}\ \lfloor\boldsymbol{\varepsilon}\rceil$, *and* $\forall C : (\Gamma^{\varphi}; T^{p}\ \lfloor\boldsymbol{\varepsilon}\rceil) \Rightarrow (\varnothing; \mathsf{Unit}^{\varnothing}\ \lfloor\boldsymbol{\varnothing}\rceil).\ C[t_1] \downarrow \iff C[t_2] \downarrow.$

We write $t \downarrow$ to mean term t terminates, if $t,\ \varnothing,\ \varnothing \Downarrow v,\ \sigma$, for some value v and final store σ.

The above definition is standard [Ahmed et al. 2009] and defines a partial program equivalence. Since we focus on a total fragment of the systems here, program termination can not be used as an observer for program equivalence. We thus rely on a refined definition of contextual equivalence using Boolean contexts, which is defined in the extended version of this paper [Bao et al. 2025].

4.2 The World Model

Following other prior works [Ahmed 2004; Benton et al. 2007; Thamsborg and Birkedal 2011], we apply Kripke logical relations to our systems. Our logical relations are indexed by types and store layouts via *worlds*, generalized store typings relating two stores. This allows us to interpret Ref T^{q} as an allocated location that holds values of type T^{q}. The invariant that all allocated locations hold well-typed values *w.r.t.* the world must hold in the pre-state and be re-established in the post-state of a computation. The world may grow as more locations may be allocated. It is important that this invariant must hold in future worlds, which is commonly referred to as *monotonicity*.

World, Relational Worlds & World Extension. As discussed in Section 3.5, our store layout is specified by the logical interpretation of reachability qualifiers in a store typing, and is free of cycles. The notion of world for our systems is defined as:

Definition 4.2 (World). *A world W is a tuple* $(\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$, *where*

- Σ_1 *and* Σ_2 *are store typings defined in Fig. 9,*
- $f \subseteq (\mathrm{dom}(\Sigma_1) \times \mathrm{dom}(\Sigma_2))$ *is a partial bijection.*
- $\hat{\mathbb{V}} \subseteq (\mathrm{dom}(\Sigma_1) \times \mathrm{dom}(\Sigma_2) \times \mathbb{V} \times \mathbb{V})$.

A world is meant to define relational stores. The partial bijection captures the fact that a relation holds under permutation of store locations. The notation $\hat{\mathbb{V}}$ maps from pairs of locations to semantic types. A semantic type is a set of values ($\mathbb{V}$).

If $\mathrm{W} = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$ is a world, we refer to its components as follows:

$$
\mathrm{W}(\ell_1, \ell_2) = \begin{cases} (\ell_1, \ell_2) \in f & \ell_1 \in \mathrm{dom}(\Sigma_1) \text{ and } \ell_2 \in \mathrm{dom}(\Sigma_2) \text{ and when defined} \\ \bot & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathrm{dom}_1(\mathrm{W}) &= \mathrm{dom}(\Sigma_1) & \mathrm{dom}_2(\mathrm{W}) &= \mathrm{dom}(\Sigma_2) & \mathrm{dom}(\mathrm{W}) &= (\mathrm{dom}(\Sigma_1), \mathrm{dom}(\Sigma_2)) \\
\mathrm{W}_1 &= \Sigma_1 & \mathrm{W}_2 &= \Sigma_2 & \mathrm{W}_{\mathbb{V}} &= \hat{\mathbb{V}}
\end{aligned}
$$

Binary Value Interpretation of Types and Terms $\boxed{\lambda^{\blacklozenge}/\lambda^{\blacklozenge}_{\varepsilon}}$

$$
\begin{aligned}
\mathcal{V}[[Bool, Bool]] &= \{(\hat{H},\ \mathrm{W}, c, c)\}\\
\mathcal{V}[[\mathsf{Ref}\ T_1^{q_1}, \mathsf{Ref}\ T_2^{q_2}]] &= \{(\hat{H},\ \mathrm{W}, \ell_1, \ell_2) \mid \mathrm{W}(\ell_1, \ell_2)\ \wedge\ \textcircled{1}\ \mathrm{WR}(\mathrm{W}, \{\ell_1\}, \{\ell_2\})\ \wedge (\mathrm{W}(\ell_i) = [[q_i]]_{\hat{H}_i} \wedge\\
&\quad \mathrm{W}_i(\ell_i)^*_{\mathrm{W}_i} \subseteq \ell_i{}^*_{\mathrm{W}_i} \wedge (\forall\, \sigma_1', \sigma_2', \mathrm{W}', v_1, v_2.\ (\sigma_1', \sigma_2') : \mathrm{W}' \wedge \mathrm{W} \equiv_{(\mathrm{W}_1(\ell_1)^*_{\mathrm{W}_1}, \mathrm{W}_2(\ell_2)^*_{\mathrm{W}_2})} \mathrm{W}' \wedge\\
&\quad L(v_i)^*_{\mathrm{W}_i'} \subseteq \mathrm{W}_i(\ell_i)^*_{\mathrm{W}_i'}\ \wedge\ \textcircled{2}\ \mathrm{WR}(\mathrm{W}', L(\sigma_1'(\ell_1))^*_{\mathrm{W}_1'}, L(\sigma_2'(\ell_2))^*_{\mathrm{W}_2'}) \Rightarrow\\
&\qquad ((v_1, v_2) \in \mathrm{W}_{\mathbb{V}}(\ell_1, \ell_2) \iff (\hat{H},\ \mathrm{W}', v_1, v_2) \in \mathcal{V}[[T_1, T_2]])))\}\\
\\
\mathcal{V}[[(x : T_1^{s_1}) \overset{\varepsilon_1}{\to} U_1^{r_1},\ &= \{(\hat{H},\ \mathrm{W}, \langle H_1,\ (\lambda x.t_1)^{q_1}\rangle, \langle H_2,\ (\lambda x.t_2)^{q_2}\rangle) \mid \textcircled{3}\ \mathrm{WR}(\mathrm{W}, [[q_1]]^{\mathrm{W}_1*}_{H_1}, [[q_2]]^{\mathrm{W}_2*}_{H_2}) \wedge\\
(x : T_2^{s_2}) \overset{\varepsilon_2}{\to} U_2^{r_2}]] &\quad (\forall v_1, v_2, \mathrm{W}', \sigma_1', \sigma_2'.\ (\sigma_1', \sigma_2') : \mathrm{W}'\ \wedge\ \mathrm{W} \equiv_{([[q_1]]^{\mathrm{W}_1*}_{H_1'}, [[q_2]]^{\mathrm{W}_2*}_{H_2'})} W' \wedge\\
&\quad \textcircled{4}\ \mathrm{WR}(\mathrm{W}', \mathrm{dom}_1(\mathrm{W}'), \mathrm{dom}_2(\mathrm{W}'))\ \wedge\ (\hat{H},\ \mathrm{W}', v_1, v_2) \in \mathcal{V}[[T_1, T_2]] \wedge\\
&\quad L(v_i)^*_{\mathrm{W}_i'} \subseteq [[\tilde{s}_i]]^{\mathrm{W}_i'*}_{\hat{H}_i} \cup_{\spadesuit \in s_i} [[q_i]]^{H_i*}_{\mathrm{W}_i'} \cup_{\blacklozenge \in s_i} \overline{[[q_i]]^{H_i*}_{\mathrm{W}_i'}} \Rightarrow\\
&\qquad (\exists\, \mathrm{W}'', \sigma_1'', \sigma_2'', v_1', v_2'.\ t_1,\ H_1; (x, v_1),\ \sigma_1' \Downarrow v_1',\ \sigma_1'' \wedge t_2,\ H_2; (x, v_2),\ \sigma_2' \Downarrow v_2',\ \sigma_2'' \wedge\\
&\qquad\quad \mathrm{W}' \sqsubseteq \mathrm{W}''\ \wedge\ \mathrm{WR}(\mathrm{W}'', \mathrm{dom}_1(\mathrm{W}''), \mathrm{dom}_2(\mathrm{W}''))\ \wedge\ (\sigma_1'', \sigma_2'') : \mathrm{W}'' \wedge\\
&\qquad\quad (\hat{H},\ \mathrm{W}'', v_1', v_2') \in \mathcal{V}[[U_1, U_2]] \wedge\\
&\qquad\quad L(v_i')^*_{\mathrm{W}_i''} \subseteq ([[\tilde{r}_i \backslash x]]^{\hat{H}_i*}_{\mathrm{W}_i''} \cap [[q_i]]^{H_i*}_{\mathrm{W}_i''}) \cup_{x \in r_i} L(v_i')^*_{\mathrm{W}_i''} \cup_{\spadesuit \in r_i} [[q_i]]^{H_i*}_{\mathrm{W}_i''}\\
&\qquad\qquad \cup_{\blacklozenge \in r_i} \overline{\mathrm{dom}(\mathrm{W}_i')} \wedge\\
&\qquad\quad \sigma_i' \xrightarrow{[[q_i]]^{\mathrm{W}_i'*}_{H_i} \cup L(v_i)^*_{\mathrm{W}_i'}} \sigma_i'' \qquad \sigma_i' \xrightarrow{[[\varepsilon_i \backslash x]]^{\mathrm{W}_i'*}_{H_i} \cup_{x \in \varepsilon_i} v_i{}^*_{\mathrm{W}_i'} \cup_{\spadesuit \in \varepsilon_i} [[q_i]]^{H_i'*}_{\mathrm{W}_i'}} \sigma_i''\ ))\ \}\\
\\
\mathcal{E}[[T^p\ \varepsilon]]_\varphi &= \{(\hat{H},\ \mathrm{W}, t_1, t_2) \mid \forall\, \sigma_1, \sigma_2.(\sigma_1, \sigma_2) : \mathrm{W}\ \wedge\ \exists\, \mathrm{W}', \sigma_1', \sigma_2', v_1, v_2.\ t_1,\ \hat{H}_1,\ \sigma_1 \Downarrow v_1,\ \sigma_1' \wedge\\
&\quad t_2,\ \hat{H}_2,\ \sigma_2 \Downarrow v_2,\ \sigma_2' \wedge \mathrm{W} \sqsubseteq \mathrm{W}'\ \wedge\ \mathrm{WR}(\mathrm{W}', \mathrm{dom}_1(\mathrm{W}'), \mathrm{dom}_2(\mathrm{W}'))\ \wedge\ (\sigma_1', \sigma_2') : \mathrm{W}' \wedge\\
&\quad (\hat{H},\ \mathrm{W}', v_1, v_2) \in \mathcal{V}[[T, T]]\ \wedge\ L(v_i)^*_{\mathrm{W}_i'} \subseteq [[\varphi \cap p]]^{\mathrm{W}_i*}_{\hat{H}_i} \cup_{\blacklozenge \in p} \overline{\mathrm{dom}(\mathrm{W}_i)} \wedge\\
&\quad (\sigma_1 \xrightarrow{[[\varphi]]^{\mathrm{W}_1*}_{\hat{H}_1}} \sigma_1'\ \wedge\ \sigma_2 \xrightarrow{[[\varphi]]^{\mathrm{W}_2*}_{\hat{H}_2}} \sigma_2') \qquad (\sigma_1 \xrightarrow{[[\varphi \cap \varepsilon]]^{\mathrm{W}_1*}_{\hat{H}_1}} \sigma_1'\ \wedge\ \sigma_2 \xrightarrow{[[\varphi \cap \varepsilon]]^{\mathrm{W}_2*}_{\hat{H}_2}} \sigma_2')\ \}\\
\\
G[[\Gamma^\varphi]] &= \{(\hat{H}, \mathrm{W})\ \mid\ \mathrm{dom}_i(\hat{H}) = \mathrm{dom}(\Gamma)\ \wedge\ \varphi \subseteq \mathrm{dom}(\Gamma)\ \wedge\ \mathrm{Inv}_o(\Gamma, \sigma, \hat{H}, \varphi)\ \wedge\ \mathrm{Inv}_S(\mathrm{W}, \hat{H}, \varphi)\}\\
\mathrm{Inv}_o(\Gamma, \mathrm{W}, \hat{H}, \varphi) &= \forall\, x, T, p.\ \Gamma(x) = T^p \Rightarrow \mathsf{WF}(p_x^\Gamma)\ \wedge\ (x \in \varphi \Rightarrow (\hat{H},\ \mathrm{W}, \hat{H}_1(x), \hat{H}_2(x)) \in \mathcal{V}[[T, T]]) \wedge\\
&\quad (\blacklozenge \notin p\ \wedge\ \tilde{p} \subseteq \varphi\ \wedge\ x \in \varphi \Rightarrow L(\hat{H}_i(x))^*_{\mathrm{W}_i} \subseteq [[\tilde{p}]]^{\hat{H}_i*}_{\mathrm{W}_i})\\
\mathrm{Inv}_S(\mathrm{W}, \hat{H}, \varphi) &= \forall p, p'.\ \tilde{p} \subseteq \varphi\ \wedge\ \tilde{p'} \subseteq \varphi\ \wedge\ \tilde{p}{*}{\cap}\tilde{p'}{*} \subseteq \varphi \Rightarrow [[\tilde{p}]]^{\hat{H}_i*}_{\mathrm{W}_i} \cap [[\tilde{p'}]]^{\hat{H}_i*}_{\mathrm{W}_i} \subseteq [[\tilde{p}{*}{\cap}\tilde{p'}{*}]]^{\hat{H}_i*}_{\mathrm{W}_i}.
\end{aligned}
$$

Fig. 10. The binary interpretation of types and terms, and semantic context interpretation. The highlighted formulas in teal are assertions on well-formed relations, and related values. Formulas in pink and in gray are exclusively for $\lambda^{\blacklozenge}$ and $\lambda^{\blacklozenge}_{\varepsilon}$.

If W and W′ are worlds, such that $\mathrm{dom}_1(\mathrm{W}) \cap \mathrm{dom}_1(\mathrm{W}') = \mathrm{dom}_2(\mathrm{W}) \cap \mathrm{dom}_2(\mathrm{W}') = \emptyset$, then W and W′ are called disjoint, and we write $\mathrm{W};\mathrm{W}'$ to mean extending W with a disjoint world W′. Let σ_1 and σ_2 be two stores. We write $(\sigma_1, \sigma_2) : \mathrm{W}$ to mean the stores are well-defined *w.r.t.* the world W, which is formally defined in Fig. 9.

Similar to the unary versions, we define relational worlds, written as $\mathrm{W} \equiv_{(\mathbb{L},\mathbb{L}')} \mathrm{W}'$ to mean that types and relations are preserved over two related worlds at a pair of locations $(\mathbb{L}, \mathbb{L}')$. Note that the pair of locations can not be arbitrary. Consider related locations at world W, *e.g.*, $(\ell_1, \ell_2) \in f$. If $\ell_1 \in \mathbb{L}$, but $\ell_2 \notin \mathbb{L}'$, then the relation is broken, resulting in ill-defined relational worlds. Thus, we use the predicate WR (defined in Fig. 9) to make sure the relation is defined in both directions. In other words, ℓ_1 must be related to some location, and it cannot be related to more than one location. Fig. 9 summarizes the definitions of these notations. We write $\mathrm{W} \sqsubseteq \mathrm{W}'$ to mean $\mathrm{W} \equiv_{(\mathrm{dom}_1(\mathrm{W}),\mathrm{dom}_2(\mathrm{W}))} \mathrm{W}'$. The definition of world extension satisfies reflexivity and transitivity.

4.3 Binary Logical Relations for $\lambda^{\blacklozenge}$ and $\lambda^{\blacklozenge}_{\varepsilon}$

This section presents the definition of binary logical relations for our systems, as shown in Fig. 10. In the figure, we use the subscript i in formulas to refer to both programs when the context is clear, where $i = 1$ or 2. For example, in the value interpretation of reference types, we write $W(\ell_i) = [\![q_i]\!]_{\hat{H}_i}$ to mean $W(\ell_1) = [\![q_1]\!]_{\hat{H}_1} \wedge W(\ell_2) = [\![q_2]\!]_{\hat{H}_2}$.

In the figure, the highlighted formulas in teal are assertions on well-formed relations, and related values. Formulas in pink and in gray are exclusively for $\lambda^{\blacklozenge}$ and $\lambda^{\blacklozenge}_{\varepsilon}$ respectively. As in their unary counterparts, $\lambda^{\blacklozenge}$ interprets observations as an approximation of write effects, and $\lambda^{\blacklozenge}_{\varepsilon}$ establishes a stronger value preservation property with explicit write effect annotations.

The Binary Value and Term Interpretation. As our systems use dependent types, we consider two related values at two equivalent but not identical types. For example, in the proof of rule β-EQUIV in Section 5.2, we relate two types across substitutions. Thus, our relational interpretation of type T is written as $\mathcal{V}[\![T, T]\!]$, which is a set of tuples of form $(\hat{H}, W, v_1, v_2)$, where $\hat{H}$ is a pair of relational value environments (defined in Fig. 10, bottom), W is a world, and v_1 and v_2 are values.

A pair of Boolean values are related if they are both true or false. A pair of locations are related if they store related values, in addition to their unary counterpart. Two closure records are related if applying their enclosing λ terms with argument values that are related at the domain type, their reduction results are related at the codomain type. In addition to the unary counterpart, we use predicate WR to ensure that the relations on the locations reachable from related values are well-defined, *e.g.*, ①, ②, ③ and ④ in Fig. 10.

Two terms are related if their reduction results are related at the given types.

Example: Re-ordering (part 1). In the following example, we assume three distinct Boolean references c_1, c_2 and c_3. The two sub-programs t_1 and t_2 update references c_1 and c_2 respectively, and are typed in the $\lambda^{\blacklozenge}_{\varepsilon}$ system that tracks observable write effects: [10]

$$\Gamma^{c_1,c_2,c_3} \vdash c_1 : (\text{Ref } \mathit{Bool})^{c_1} \quad \Gamma^{c_1,c_2,c_3} \vdash c_2 : (\text{Ref } \mathit{Bool})^{c_2} \quad \Gamma^{c_1,c_2,c_3} \vdash c_3 : (\text{Ref } \mathit{Bool})^{c_3}$$

$$t_1 \overset{\text{def}}{=} c_1 := !c_3 \wedge !c_1 \qquad \Gamma^{c_1,c_3} \vdash t_1 : \mathit{Bool}\ c_1$$

$$t_2 \overset{\text{def}}{=} c_2 := !c_3 \wedge !c_2 \qquad \Gamma^{c_2,c_3} \vdash t_2 : \mathit{Bool}\ c_2$$

The goal is to show that $t_1; t_2$ and $t_2; t_1$ are logically equivalent: $\Gamma^{c_1,c_2,c_3} \models t_1; t_2 \approx_{\log} t_2; t_1 : \mathit{Bool}\ c_1, c_2$. Let $(\hat{H}, W) \in G[\![\Gamma^{c_1,c_2,c_3}]\!]$. By the definition of typing context interpretation in Fig. 10, we know that $(\hat{H}, W, \hat{H}_1(c_i), \hat{H}_2(c_i)) \in \mathcal{V}[\![\text{Ref } \mathit{Bool}, \text{Ref } \mathit{Bool}]\!]$, where $i \in \{1, 2, 3\}$. By the definition of value interpretation of reference types, we know that there exists three pairs of locations (referred by the three references), which are related in the world W (formula ①). Thus, each pair of the related locations stores a pair of equivalent Boolean values (formula ② in Fig. 10), where all the referents (boolean values) reach the empty location, satisfying the premise of formula ② trivially. We can prove the equivalence by applying Theorem 5.1 in Section 5.1.

The Time Travelling Property. Similar to the unary versions, our relational reasoning can time travel back and forth from a past world to another (possible future) one as long as they agree on pairs of reachable locations from the given values:

Lemma 4.3 (Time Travelling for Binaries). *If* $(\sigma_1, \sigma_2) : W$, *and* $(\hat{H}, W, v_1, v_2) \in \mathcal{V}[\![T, T']\!]$, *and* $W \equiv_{(L(v_1)^*_{W_1}, L(v_2)^*_{W_2})} W'$, *then* $(\hat{H}, W', v_1, v_2) \in \mathcal{V}[\![T, T']\!]$.

This lemma allows us to prove the validity of a pair of values preserved *w.r.t.* a constructed world in the proof of rules RE-ORDER-$\lambda^{\blacklozenge}$, RE-ORDER-$\lambda^{\blacklozenge}_{\varepsilon}$ and β-EQUIV in Section 5.

4.4 The Compatibility Lemmas, Fundamental Theorem and Soundness

Fig. 10 (bottom) defines the interpretation of typing contexts. In the definition, $\hat{H}$ ranges over a pair of relational value environments that are finite maps from variables x to pairs of values (v_1, v_2). If $\hat{H}(x) = (v_1, v_2)$, then $\hat{H}_1(x)$ denotes v_1 and $\hat{H}_2(x)$ denotes v_2. We write $\text{dom}_1(\hat{H})$ and $\text{dom}_2(\hat{H})$ to

[10] Conjunction can be encoded by our sequence terms.

Equational Rules $\boxed{\lambda^{\blacklozenge}/\lambda^{\blacklozenge}_{\varepsilon}}$

$$(\textsc{re-order-}\lambda^{\blacklozenge})\ \frac{\Gamma^{\varphi_1} \vdash t_1 : Bool^{p_1} \quad \Gamma^{\varphi_2} \vdash t_2 : Bool^{p_2} \quad \varphi_1 \subseteq \varphi \quad \varphi_2 \subseteq \varphi \quad \varphi_1{*} \cap \varphi_2{*} = \varnothing}{\Gamma^{\varphi} \models t_1; t_2 \approx_{\log} t_2; t_1 : Bool^{\varnothing}}$$

$$(\textsc{re-order-}\lambda^{\blacklozenge}_{\varepsilon})\ \frac{\Gamma^{\varphi_1} \vdash t_1 : Bool^{p_1}\ \varepsilon_1 \quad \Gamma^{\varphi_2} \vdash t_2 : Bool^{p_2}\ \varepsilon_2 \quad \varphi_1 \subseteq \varphi \quad \varphi_2 \subseteq \varphi \quad \varphi_1{*} \cap \varepsilon_2{*} = \varnothing \quad \varphi_2{*} \cap \varepsilon_1{*} = \varnothing}{\Gamma^{\varphi} \models t_1; t_2 \approx_{\log} t_2; t_1 : Bool^{\varnothing}\ \varepsilon_1 \rhd \varepsilon_2}$$

$$(\beta\textsc{-equiv})\ \frac{\begin{array}{c}(\Gamma,\ x : T^{\varphi \cap (s[q/\diamondsuit]) \cup_{\blacklozenge \in s} \blacklozenge})^{q',x} \vdash t_2 : U^{r[q'/\diamondsuit]}\ \varepsilon[q'/\diamondsuit] \qquad \Gamma^{\varnothing} \vdash t_1 : T^{\varnothing}\ \varnothing \\ q' = \varphi \cap q \qquad \tilde{s} \subseteq q' \qquad \theta = [\varnothing/x, q/\diamondsuit]\end{array}}{\Gamma^{\varphi} \models (\lambda x.t_2)^{q'}\ t_1 \approx_{\log} t_2[t_1/x] : U^{r\theta}\ \varepsilon\theta}$$

Fig. 11. The re-ordering and β-inlining rules for $\lambda^{\blacklozenge}/\lambda^{\blacklozenge}_{\varepsilon}$.

mean the domain of the first and second value environment respectively. The binary semantic typing judgment is defined as: $\Gamma^{\varphi} \models t_1 \approx_{\log} t_2 : T^{p}\ \llcorner\varepsilon\lrcorner \overset{\text{def}}{=} \forall(\hat{H}, \mathrm{W}) \in G[[\Gamma^{\varphi}]].(\hat{H}, \mathrm{W}, t_1, t_2) \in \mathcal{E}[[T^{p}\ \llcorner\varepsilon\lrcorner]]_{\varphi}$.

Theorem 4.4 (Fundamental Property). *If* $\Gamma^{\varphi} \vdash t : T^{p}\ \llcorner\varepsilon\lrcorner$, *then* $\Gamma^{\varphi} \models t \approx_{log} t : T^{p}\ \llcorner\varepsilon\lrcorner$.

Lemma 4.5 (Congruency of Binary Logical Relations). *The binary logical relation is closed under well-typed program contexts,* i.e., *if* $\Gamma^{\varphi} \models t_1 \approx_{log} t_2 : T^{p}\ \llcorner\varepsilon\lrcorner$, *and* $C : (\Gamma^{\varphi}; T^{p}\ \llcorner\varepsilon\lrcorner) \Rrightarrow (\Gamma'^{\varphi'}; T'^{p'}\ \llcorner\varepsilon'\lrcorner)$, *then* $\Gamma'^{\varphi'} \models C[t_1] \approx_{log} C[t_2] : T'^{p'}\ \llcorner\varepsilon'\lrcorner$.

Lemma 4.6 (Adequacy of the binary logical relations). *The binary logical relation preserves termination,* i.e., *if* $\varnothing \models t_1 \approx_{log} t_2 : T^{\varnothing}\ \llcorner\varnothing\lrcorner$, *then* $\exists\ \sigma, \sigma', v_1, v_2.\ t_1,\ \varnothing,\ \varnothing \Downarrow v_1,\ \sigma \wedge t_2,\ \varnothing,\ \varnothing \Downarrow v_2,\ \sigma'$.

Theorem 4.7 (Soundness of Binary Logical Relations). *The binary logical relation is sound w.r.t. contextually equivalence,* i.e., *if* $\Gamma^{\varphi} \vdash t_1 : T^{p}\ \llcorner\varepsilon\lrcorner$ *and* $\Gamma^{\varphi} \vdash t_2 : T^{p}\ \llcorner\varepsilon\lrcorner$, *then* $\Gamma^{\varphi} \models t_1 \approx_{log} t_2 : T^{p}\ \llcorner\varepsilon\lrcorner$ *implies* $\Gamma^{\varphi} \models t_1 \approx_{ctx} t_2 : T^{p}\ \llcorner\varepsilon\lrcorner$.

The formalization of the encapsulated computation lemma for the binary logical relations is omitted here. We will show its use in the proof of rule β-equiv (*i.e.*, Lemma 5.2) in Section 5.2.

Discussion: Example in Fig. 3. Now we provide informal justification for the safe use of unsafe features (*i.e.*, the assertion statement) in the example of Fig. 3, using the binary logical relation for $\lambda^{\blacklozenge}_{\varepsilon}$. We reduce the problem by showing the program where the assertion is substituted with `!x == 1` is observationally equivalent to the one where the assertion is substituted with `true`. This allows us to establish the safety of the assertion without extending $\lambda^{\blacklozenge}_{\varepsilon}$. The reasoning is straightforward: by the fundamental property, the two programs right before the substituted statements are observationally equivalent, where their two local references are related. Moreover, since the effect of function `f` is variable y, which reaches locations separate from variable x. Thus, the referent of x remains unchanged, and hence, the two programs are observationally equivalent.

5 Equational Theory

This section shows applications of our binary logical relations by proving two re-ordering and β-equivalence rules. We outline the key ideas of the proofs, primarily by presenting key lemmas and suitable worlds definitions used in the proof. Full proofs can be found in the Rocq artifacts.

5.1 Re-ordering

Rules re-order-$\lambda^{\blacklozenge}$ and re-order-$\lambda^{\blacklozenge}_{\varepsilon}$ in Fig. 11 permit two computations to commute. The former is applied to $\lambda^{\blacklozenge}$, and requires the observations of the two terms to be separate; the latter is applied to $\lambda^{\blacklozenge}_{\varepsilon}$, and demands the observation of one term to be separate from the effects of the other. Here we outline the proof of rule re-order-$\lambda^{\blacklozenge}_{\varepsilon}$. Rule re-order-$\lambda^{\blacklozenge}$ can be proved analogously.

In Fig. 12, the left side shows the definitions of the initial world W (top) and final world W^{R} (bottom), the semantics of programs $\mathrm{Prog}_1 = t_1; t_2$ and $\mathrm{Prog}_2 = t_2; t_1$ (following the two vertical

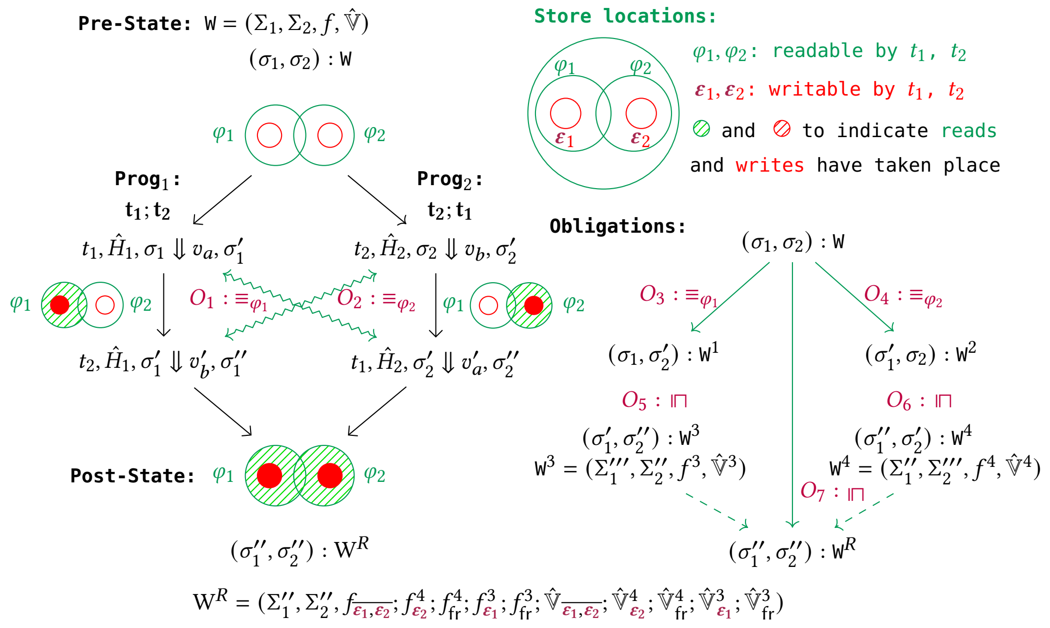
Pre-State: $W = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$
$(\sigma_1, \sigma_2) : W$
φ_1
φ_2
Prog$_1$: $t_1; t_2$
Prog$_2$: $t_2; t_1$
$t_1, \hat{H}_1, \sigma_1 \Downarrow v_a, \sigma_1'$
$t_2, \hat{H}_2, \sigma_2 \Downarrow v_b, \sigma_2'$
$O_1 : \equiv_{\varphi_1}$
$O_2 : \equiv_{\varphi_2}$
$t_2, \hat{H}_1, \sigma_1' \Downarrow v_b', \sigma_1''$
$t_1, \hat{H}_2, \sigma_2' \Downarrow v_a', \sigma_2''$
Post-State:
$(\sigma_1'', \sigma_2'') : W^R$
Store locations:
φ_1, φ_2: readable by t_1, t_2
$\varepsilon_1, \varepsilon_2$: writable by t_1, t_2
▨ and ▨ to indicate reads and writes have taken place
ε_1
ε_2
Obligations:
$(\sigma_1, \sigma_2) : W$
$O_3 : \equiv_{\varphi_1}$
$O_4 : \equiv_{\varphi_2}$
$(\sigma_1, \sigma_2') : W^1$
$(\sigma_1', \sigma_2) : W^2$
$O_5 : \sqsubseteq$
$O_6 : \sqsubseteq$
$(\sigma_1', \sigma_2'') : W^3$
$W^3 = (\Sigma_1''', \Sigma_2'', f^3, \hat{\mathbb{V}}^3)$
$(\sigma_1'', \sigma_2') : W^4$
$W^4 = (\Sigma_1'', \Sigma_2''', f^4, \hat{\mathbb{V}}^4)$
$O_7 : \sqsubseteq$
$(\sigma_1'', \sigma_2'') : W^R$
$W^R = (\Sigma_1'', \Sigma_2'', f_{\overline{\varepsilon_1, \varepsilon_2}}; f^4_{\varepsilon_2}; f^4_{\mathrm{fr}}; f^3_{\varepsilon_1}; f^3_{\mathrm{fr}}; \hat{\mathbb{V}}_{\overline{\varepsilon_1, \varepsilon_2}}; \hat{\mathbb{V}}^4_{\varepsilon_2}; \hat{\mathbb{V}}^4_{\mathrm{fr}}; \hat{\mathbb{V}}^3_{\varepsilon_1}; \hat{\mathbb{V}}^3_{\mathrm{fr}})$

Fig. 12. Illustrated proof that $\mathrm{Prog}_1 = t_1; t_2$ and $\mathrm{Prog}_2 = t_2; t_1$ are logically equivalent at world W. The key obligations are (1) establishing equivalence of subterms (t_1, t_1) and (t_2, t_2) at worlds W^1 and W^2 respectively (O_1 & O_2); and (2) showing the final store pairs (σ_1'', σ_2'') are well-formed *w.r.t.* world W^R, an extension of the inital one W (O_7).

arrows), and the major proof objections ($O_1 - O_7$). The bottom right shows world relations. We write $(\sigma_1, \sigma_2) : W \equiv_{\varphi_1} (\sigma_1, \sigma_2') : W^1$ to mean that stores (σ_1, σ_2) and (σ_1, σ_2') are well-formed *w.r.t.* worlds W and W^1 respectively, and worlds W^1 and W are related *w.r.t.* to φ_1. The notation $(\sigma_1, \sigma_2') : W^1 \sqsubseteq (\sigma_1', \sigma_2'') : W^3$ is defined similarly, but for world extension. We need to show that programs Prog_1 and Prog_2 are logically equivalent at world W, where $(\hat{H}, W) \in G[[\Gamma^{\varphi}]]$.

As shown at the top right of Fig. 12, the entire set of store locations consists of three parts: those not observed by either the execution of term t_1 or t_2, *i.e.*, $\overline{\varphi_1, \varphi_2}$, and those needed by the execution of term t_1 and t_2 respectively, *i.e.*, φ_1 and φ_2. In addition, t_1 and t_2 exclusively modify locations in ε_1 and ε_2, respectively.

Key Idea: Term Equivalence Preservation (Obligations 1 & 2). From the fundamental property (Theorem 4.4), we know that $(t_2, t_2)/(t_1, t_1)$ are logically equivalent at world W. Observing the semantics in Fig. 12, term t_1/t_2 is evaluated first in $\mathrm{Prog}_1/\mathrm{Prog}_2$. The key is to show that after evaluating t_1/t_2, the equivalence $(t_2, t_2)/(t_1, t_1)$ still holds at some world. Informally, the observation $\varphi_2/\ \varphi_1$ specifies what is needed for maintaining the equivalence of $(t_2, t_2)/(t_1, t_1)$ at world W respectively. As the observation of one term is separate from the effects of the other, their equivalences are preserved. This property is formalized in the extended version of this paper [Bao et al. 2025].

Well-Formed Relational Worlds (Obligations 3 & 4). We know $W \sqsubseteq_{\varphi_2} W^2$ and $W \sqsubseteq_{\varphi_1} W^1$ by the definition of relational worlds in Fig. 10.

Well-Formed World Extensions (Obligations 5 & 6). These can be discharged by the definition of term interpretation in Fig. 10.

Well-Formed Final World (Obligation 7). We discuss the key technical points that show world W^R is a valid extension of the initial world W, and the pair of final stores are well-formed *w.r.t.* W^R.

We first introduce the notations $f_{\overline{\varepsilon_1,\varepsilon_2}}$, $f^4_{\varepsilon_2}$, and $f^3_{\varepsilon_1}$ shown in Fig. 12. Without loss of generalization, we assume $(\hat{H}, \mathrm{W}) \in G[[\Gamma^{\varphi}]]$, and $\mathrm{W} = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$, and the effect $\boldsymbol{\varepsilon}$ is closed at Γ. The notation f_{ε} denotes that weakening the relation to the part only covered by the interpreation of $\boldsymbol{\varepsilon}$, *i.e.*, $f_{\varepsilon} = \{(\ell_1, \ell_2).\ (\ell_1, \ell_2) \in f \wedge \ell_1 \in [[\varphi \cap \boldsymbol{\varepsilon}]]^{\hat{H}_1 *}_{\mathrm{W}_1} \wedge \ell_2 \in [[\varphi \cap \boldsymbol{\varepsilon}]]^{\hat{H}_2 *}_{\mathrm{W}_2}\}$. For example, $\mathrm{W}_{\overline{\varepsilon_1;\varepsilon_2}}$ denotes the relations that are unchanged during the computation.

Now we show that $f^3_{\varepsilon_1} = f_{\varepsilon_1}$ and $f^4_{\varepsilon_2} = f_{\varepsilon_2}$. This is established by $\mathrm{W} \equiv_{\varphi_2} \mathrm{W}^2$ and $\mathrm{W}^2 \sqsubseteq \mathrm{W}^4$, *i.e.*, W^1 preserves the relation from f_{φ_2}, and W^4 is a valid extension of W^2. The notation f^3_{fr} means the related locations that are fresh at world W^3, *i.e.*, $\forall \ell_1, \ell_2, f^3(\ell_1, \ell_2) \wedge \ell_1 \in \mathsf{fr}(\mathrm{W}^3_1, \mathrm{W}^1_1) \wedge \ell_2 \in \mathsf{fr}(\mathrm{W}^3_2, \mathrm{W}^1_2)$. The meaning of the notation f^4_{fr} is defined similarly. Now, we can see that we construct the relation in a mutually disjoint way, *i.e.*, W^R_f and $\mathrm{W}^R_{\hat{\mathbb{V}}}$ are well-defined, and world W^R is a valid extension of W. From Fig. 12, we know that $\sigma''_1 : \mathrm{W}^4_1$ and $\sigma''_2 : \mathrm{W}^3_2$. Thus, the pair of final stores are well-formed *w.r.t.* world W_R.

Theorem 5.1. *Rules* RE-ORDERING-$\lambda^{\blacklozenge}$ *and* RE-ORDERING-$\lambda^{\blacklozenge}_{\varepsilon}$ *in Fig. 11 are sound.*

Example: Re-ordering (part 2). The re-ordering example in Section 4.3 can now be proved directly by applying Theorem 5.1, where $\varphi_1 = c_1, c_3$, $\varphi_2 = c_2, c_3$, $\varphi = c_1, c_2, c_3$, $\boldsymbol{\varepsilon_1} = c_1$, and $\boldsymbol{\varepsilon_2} = c_2$, and the side conditions of rule RE-ORDER-$\lambda^{\blacklozenge}_{\varepsilon}$ are all satisfied.

Discussion: Stronger than Parallel Reduction. Prior works on reachability types [Bao et al. 2021; Wei et al. 2024] do not consider any notion of equivalance. The "progress and preservation in parallel reduction" property (Corollary 4.9) in prior work [Wei et al. 2024] only shows that reduction can proceed in parallel for top-level programs, but does not show that commuting two reduction sequences yields equivalent results in arbitrary typed program contexts. Thus, the corollary in their work cannot be used to justify our re-ordering rules.

5.2 β-Equivalence

Rule β-EQUIV in Fig. 11 permits replacing a function call site t_1 with the body of the called function, provided that t_1 is observably pure. Let Prog_1 be $(\lambda x.t_2)^{q'}\, t_1$, and Prog_2 be $t_2[t_1/x]$. We need to show that programs Prog_1 and Prog_2 are logically equivalent at world W, where $(\hat{H}, \mathrm{W}) \in G[[\Gamma^{\varphi}]]$, $\mathrm{W} = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$, and $(\sigma_1, \sigma_2) : \mathrm{W}$. Recall the program semantics in the followings:

$$\begin{array}{ll}\mathrm{Prog}_1: & (\lambda x.t_2)^{q'},\ \hat{H}_1,\ \sigma_1 \Downarrow \langle \hat{H}_1,\ (\lambda x.t_2)^{q'} \rangle,\ \sigma'_1 \quad t_1, \hat{H}_1, \sigma'_1 \Downarrow v_a, \sigma''_1 \quad t_2, \hat{H}_1; (x : v_a), \sigma''_1 \Downarrow v_2, \sigma'''_1 \\ \mathrm{Prog}_2: & t_2[t_1/x], \hat{H}_2, \sigma_2 \Downarrow v'_2, \sigma'_2\end{array}$$

Observing the above semantics of Prog_1, we know that the λ-term gets evaluated first with final store σ'_1, such that $\sigma'_1 : \Sigma'_1$. Then t_1 gets evaluated to value v_a with final store σ''_1, such that $\sigma''_1 : \Sigma''_1$. Let W^1 be $(\Sigma'_1, \Sigma, f, \hat{\mathbb{V}})$, and W^2 be $(\Sigma''_1, \Sigma, f, \hat{\mathbb{V}})$. By the definition of world extension in Fig. 10, we know that $\mathrm{W} \sqsubseteq \mathrm{W}^1$, and $\mathrm{W}^1 \sqsubseteq \mathrm{W}^2$. Thus, we know $\mathrm{W} \sqsubseteq \mathrm{W}^2$ by the transitivity of world extension.

Now, our goal is to prove $(\hat{H}_1; (x : v_a), \hat{H}_2, W^2, t_2, t_2[t_1/x]) \in \mathcal{E}[[U^{r\theta}\ \boldsymbol{\varepsilon\theta}]]_{\varphi}$, where $\theta = [\varnothing/x, q'/\diamondsuit]$. In Prog_2, term t_1 may get evaluated any time during the course of evaluating $t_2[t_1/x]$, where t_2's effects may already occur. Note term t_1's observation is the empty set, *i.e.*, establishing (t_1, t_1) equivalence does not need any related locations in the pre-stores. We formalize this observation in the following lemma, which is an application of encapsulated computation lemma for binaries:

Lemma 5.2. *If* $\Gamma^{\varnothing} \vdash t : T^{\varnothing}\ \boldsymbol{\varnothing}$, *and* $(\hat{H}, W) \in G[[\Gamma^{\varnothing}]]$, *and* $W = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$, *then* $(\hat{H},\ (\Sigma_1, \Sigma_2, \varnothing, \varnothing), t, t) \in \mathcal{E}[[T^{\varnothing}\ \boldsymbol{\varnothing}]]_{\varnothing}$.

From the lemma above, we know that the resulting values (v_1, v_2) reach the empty set of locations, *i.e.*, $L(v_i)^*_{\mathrm{W}^c_i}$, where W^c is the resulting world and $i = 1$ or 2. Now we know that (v_1, v_2) are logically equivalent at any world by Lemma 4.3. The following shows the top-level semantic weakening lemma that captures the fact that (1) t_1 may get evaluated any time during the course of evaluating $t_2[x/t_1]$; (2) v_a and the resulting value from t_1 in Prog_2 are logically equivalent at any given world.

Lemma 5.3 (Semantic Weakening). *If* $\Gamma^{\varnothing} \vdash t_1 : T^{\varnothing}\ \varnothing$, *and* $W^{1'} = (\Sigma_1', \Sigma_2, \varnothing, \varnothing)$, *and* $(\hat{H}, W^{1'}) \in G[[\Gamma^{\varnothing}]]$, *and* $(\sigma_1', \sigma_2) : W^{1'}$, *and there exists* v_a, σ_1'' *and* Σ_1'', *such that*

- $t_1, H_1, \sigma_1' \Downarrow v_a, \sigma_1''$, *and* $L(v_a) \subseteq \varnothing$, *and* $\sigma_1' \rightarrow_{\varnothing} \sigma_1''$, *and* $(\sigma_1'' : \Sigma_1'')$, *and* $\Sigma_1' \sqsubseteq \Sigma_1''$, *and*

(for all H_2', σ_2^b, Σ_2^b, $\mathbb{L}$. $\Sigma_2 \equiv_{\mathbb{L}} \Sigma_2^b$, *and* $(\sigma_2^b : \Sigma_2^b)$ *implies there exists* v_b', σ_2^c, Σ_2^c, *such that*

- $t_1, \hat{H}_2; H_2', \sigma_2^b \Downarrow v_b', \sigma_2^c$, *and* $\sigma_2^b \rightarrow_{\varnothing} \sigma_2^c$, *and* $\Sigma_2^b \sqsubseteq \Sigma_2^c$, *and*

$W^c = (\Sigma_1'', \Sigma_2^c, \varnothing, \varnothing)$, *and* $(\sigma_1'', \sigma_2^c) : W^c$, *and* $((\hat{H}_1, \hat{H}_2; H_2'), W^c, v_a, v_b') \in \mathcal{V}[[T, T]]$, *and* $L(v_a)^*_{\Sigma_1''} \subseteq \varnothing$, *and* $L(v_b')^*_{\Sigma_2^c} \subseteq \varnothing$*).*

The following shows the top-level semantic substitution lemma, which uses the conclusion of the semantic weakening lemma as hypothesis.

Lemma 5.4 (Semantic Substitution). *If* $(\Gamma,\ x : T^{\varphi \cap (s[q/\diamondsuit]) \cup_{\blacklozenge \in s} \blacklozenge})^{q',x} \vdash t_2 : U^{r[q'/\diamondsuit]}\ \boldsymbol{\varepsilon}[q'/\diamondsuit]$, *and* $q' = \varphi \cap q$, *and* $\tilde{S} \subseteq q'$, *and* $r \backslash x \subseteq \varphi$, *and* $\boldsymbol{\varepsilon} \backslash x \subseteq \varphi$, *and* $W = (\Sigma_1, \Sigma_2, f, \hat{\mathbb{V}})$, *and* $W^2 = (\Sigma_1'', \Sigma_2, f, \hat{\mathbb{V}})$, *and* $(\hat{H}, W^2) \in G[[\Gamma^{\varphi}]]$ *and* $WR(W, \text{dom}_1(W), \text{dom}_2(W))$ *and* $WR(W, \text{dom}_1(W^2), \text{dom}_2(W^2))$, *and* t_1 *satisfies semantic weakening (Lemma 5.3), then* $(\hat{H}_1; (x : v_a), \hat{H}_2), W^2, t_2, t_2[t_1/x] \in \mathcal{E}[[U^{r\theta}\ \boldsymbol{\varepsilon}\theta]]_{\varphi}$, *where* $\theta = [\varnothing/x, q'/\diamondsuit]$.

In the proofs of Lemma 5.3 and Lemma 5.4, we give new definitions of typing context interpretation for the not-aligned value environments, and define term substitution (omitted here). Their proofs are conducted by induction on the type derivation of t_1 and t_2 respectively.

From the conclusion of Lemma 5.4, we get resulting world W′. Let f' and $\hat{\mathbb{V}}'$ be the relation and the related values in W′ respectively. Now we construct the world as $(\Sigma_1''', \Sigma_2', f_{\overline{\varphi}}; f_{\varphi}'; f_{\text{fr}}'; \hat{\mathbb{V}}_{\overline{\varphi}}; \hat{\mathbb{V}}_{\varphi}'; \hat{\mathbb{V}}_{\text{fr}}')$, which is a valid extension of initial world W. The remainder of the proof follows similarly to the proof of reordering and is thus omitted.

Theorem 5.5. *Rule β-EQUIV in Fig. 11 is sound.*

Discussion: Extensions. Rule β-EQUIV restricts the argument t_1 to an empty observation set, empty reachability qualifier, and empty write effect. While effects certainly have to be excluded (*e.g.*, substitution may lead to duplication of t_1), scenarios with non-empty observation should also be safe, as long as t_1 is in the restricted form of variables or λ-terms. While we have not yet mechanized such extended cases in Rocq, a proof would need to show that the equivalence of two terms is preserved over relational worlds *w.r.t.* the reachable locations from the given values, and the hypothesis of Lemma 5.3 needs to change $W^{1'}$ to $(\Sigma_1', \Sigma_2, \varphi_1)$, capturing the fact that reduction of t_1 requires locations specified in φ_1. Given the experience with different reordering rules, we are confident that our binary logical relation can mirror the full range of possible application cases, including fresh/non-fresh arguments, self-references in the function type, etc.

6 Related Work

Reachability/Capturing Types. Prior works on reachability types [Bao et al. 2021; Wei et al. 2024] proved syntactic type soundness (progress and preservation). This paper presents semantic models of reachability systems and establishes stonger semantic properties via logical relations. The models in Section 3 are based on $\lambda^{\blacklozenge}$ in Wei et al. [2024]'s work, and have already been extended with support for subtyping and polymorphism in Jia et al. [2024]'s work. Recently, Deng et al. [2025] refined Wei et al. [2024]'s system to support cyclic references, addressing the key limitation identified in this paper. We leave it to future work to model cyclic store structures and non-terminating programs using step-indexed logic relations. In parallel to this work, Bunting and Murawski [2025] investigate contextual equivalence for Bao et al. [2021]'s system using operational game semantics and present a full abstraction model based on a labelled transition system.

Closely related to reachability types and also tracking variables in types, capturing types [Boruch-Gruszecki et al. 2023; Xu and Odersky 2023] are a recent effort to integrate capability tracking and escape checking into Scala 3. While the overall goals and design are similar, there are also important differences. Specifically, the use of self-references for escaping values and the modeling of freshness is unique to reachability types.

Tracking Sharing/Uniqueness/Immutability. [Giannini et al. 2019a,b] develops calculi that infer introduced sharing during an expression's execution. and can detect capsule references and borrowing. Those works adopt a non-standard operational model that encodes store in the language term. In contrast, reachability types tracks sharing as well, but uses the standard operational semantics. [Bianchini et al. 2022] tracks sharing and mutation in a coeffect system that can model uniqueness and immutability of references. Reachability types track sharing in qualifiers, and mutation is tracked via the write effect extension (see Section 3.6). Language features, *i.e.*, uniqueness and borrowing, can be enforced by a flow-sensitive effect system, which are discussed in prior works [Bao et al. 2021; Wei et al. 2024]. A recent work [Deng et al. 2025] introduces cyclic reference types with dual-component that can be used to enforce read-only references. The logical relations presented in this work does not rely on uniqueness and immutability of references.

Ownership Types. Ownership type systems [Clarke et al. 2013, 1998; Noble et al. 1998; Potanin et al. 2006] typically enforce strict heap invariants and selectively re-introduce sharing in a controlled way via borrowing [Clebsch et al. 2015a; Hogg 1991; Naden et al. 2012]. The Rust type system [Matsakis and Klock 2014] adopts a strong ownership model. Its "shared XOR mutable" mechanism enforces uniqueness of mutable references while allowing sharing among immutable ones. Marshall and Orchard [2024]'s work connects Granule's uniqueness modality [Orchard et al. 2019] and its graded model types with ideas from Rust, allowing ownership tracking in a graded type system. In contrast, reachability types are designed to track sharing (thus separation), rather than restrict it. The focus is on higher-level languages, where many common idioms rely on sets of closures as an interface to interact with a piece of (shared!) mutable state. Uniqueness properties can be enforced via flow-sensitive move effects [Bao et al. 2021; Jia et al. 2024]. In the context of Rust, Jung et al. [2018a] developed lifetime logic to model lifetimes and borrowing in a variant of Rust's MIR, where programs are expressed in continuation-passing style. Their logical framework can verify unsafe code with safe APIs for Rust programs. In contrast, our work aims to verify equational rules using observational equivalence. We showed an informal justification of a safe use of assertion statements by reducing the problem to proving observational equivalence (Section 4.4), but unlike Jung et al. [2018a]'s work, our goal is not to verify safe use of general unsafe code.

Capabilities and Permissions. Capability systems [Boyland et al. 2001; Castegren and Wrigstad 2016] have been used to reason about program resources and external calls [Drossopoulou et al. 2025]. In Pony [Clebsch 2017], reference capabilities describe what other aliases are denied [Clebsch et al. 2015b; Dodds et al. 2009], *e.g.*, deny local/global read/write aliases. In contrast, reachability types [Bao et al. 2021; Wei et al. 2024] do not distinguish read and write capabilities. Read-only capabilities can be enforced via cyclic reference types with dual-component [Deng et al. 2025], which is not modeled in this work. Haller and Odersky [2010] leverage capabilities to ensure externally unique access with borrowing. Gordon et al. [2012] extend their work with immutable references for safe parallelism. Their symmetric parallelism rule enforces constraints on read and write that are similar to our rule RE-ORDERING-$\lambda^{\blacklozenge}_{\varepsilon}$. In contrast, our system does not rely on the uniqueness or immutability of references, but instead leverages aliasing/separation and write effects tracking. Unique mutable references can be enforced via a flow-sensitive effect system, which has been discussed in prior works [Bao et al. 2021; Wei et al. 2024]. We leave a thorough investigation as future work.

Step-Indexed Logical Relations. Step-indexed logical relations have been used to prove semantic type soundness [Ahmed 2004; Appel and McAllester 2001], program termination [Spies et al. 2021; Timany et al. 2024] and contextual equivalence [Ahmed 2006], representation independence [Ahmed et al. 2009; Timany et al. 2024], monadic encapsulation of state achieved by runST using rank-2 polymorphism [Timany et al. 2018] for languages with general recursive and polymorphic types. In these approaches, types are indexed by computation steps to guarantee well-founded recursions have unique fixed points. Since all well-typed programs are guaranteed to terminate in our systems, our LR models do not require step-indexes. As future work, we plan to extend our models to support recursion and cyclic references, and we are interested in using the Iris logic framework [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017] to avoid manual step indexes and similar proof devices.

Effect-Based Program Transformation. Close to our approach that uses effects for validating program transformations, Benton et al. [2014, 2007] developed denotational, semantic relational models based on a monotonic regions and effects for a higher-order language with dynamically allocated first-order mutable stores, and with high-order stores without dynamic allocation [Benton et al. 2009, 2006]. Our world model is adapted from Benton et al. [2007]'s work, and our use of observable write effects to prove observational equivalence coincides with Benton et al. [2014]'s work. Unlike their works, we adopt a big-step operational semantics based on closures and environments, which is closer to real language implementations. In addition, our effect system does not track allocation effects, thus, our models assume allocation always occurs during the reduction of a term. This design works well, as we can leverage reachability qualifiers. For example, the absence of the freshness marker indicates values do not reach fresh locations. In this case, freshly allocated locations cannot be reached from clients. Benton et al. [2016] developed models to validate effect-dependent program transformations for concurrent programs, which is beyond the scope of this work and left as future work.

World Models. Birkedal et al. [2016]; Thamsborg and Birkedal [2011]'s work defines a Kripke logical relation based on a region-polymorphic type-and-effect system for an ML-like programming language with higher-order mutable stores with dynamic allocations. In their work, a world is the part of the entire store that the program controls, while our world describes the entire store, and we use the separation of saturated reachability qualifiers/observations and observable write effects to achieve different degrees of local reasoning. Reachability types feature lightweight reachability polymorphism via qualifier dependent function application, and use self-references to succinctly express if returned values are fresh, without introducing explicit quantification. Wei et al. [2024]'s work has proven syntactic type soundness for reachability and type polymorphism ($\mathsf{F}^{\blacklozenge}_{<:}$).

7 Conclusion

In this paper, we presented semantic models for a family of reachability type systems with increasing sets of features, and showed how reachability types naturally lead to strong local reasoning properties. The results established, *i.e.*, semantic type soundness, termination, effect safety, and program equivalence, are stronger than those in prior works on reachability types. We also proved key equational rules such as reordering and β-equivalence, providing a firm foundation for reachability-driven program transformations. All results are proven in Rocq.

Data-Availability Statement

Rocq mechanizations can be found at https://github.com/tiarkrompf/reachability and [Bao 2025].

Acknowledgments

We thank Siyuan He and Haotian Deng for related contributions to reachability types. This work was supported in part by NSF awards 2348334 and Augusta faculty startup package, as well as gifts from Meta, Google, Microsoft, and VMware.

References

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. ACM, 340–353. doi:10.1145/1480881.1480925

Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Princeton University.

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP (Lecture Notes in Computer Science, Vol. 3924)*. Springer, 69–83. doi:10.1007/11693024_6

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science, Vol. 9600)*. Springer, 249–272. doi:10.1007/978-3-319-30936-1_14

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. ACM, 666–679. doi:10.1145/3009837.3009866

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

Yuyan Bao. 2025. *Reproduction Package for Article 'Modeling Reachability Types with Logical Relations'.* doi:10.5281/zenodo.16934167

Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations: Semantic Type Soundness, Termination, and Equational Theory (Extended Version). arXiv:2309.05885 [cs.PL]

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. doi:10.1145/3485516

Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *POPL*. ACM, 619–632. doi:10.1145/2535838.2535869

Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs. In *PPDP*. ACM, 188–201. doi:10.1145/2967973.2968602

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*. ACM, 87–96. doi:10.1145/1273920.1273932

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*. ACM, 301–312. doi:10.1145/1599410.1599447

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, Writing and Relations. In *APLAS (Lecture Notes in Computer Science, Vol. 4279)*. Springer, 114–130. doi:10.1007/11924661_7

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022. Coeffects for sharing and mutation. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 870–898. doi:10.1145/3563319

Lars Birkedal, Guilhem Jaber, Filip Sieczkowski, and Jacob Thamsborg. 2016. A Kripke logical relation for effect-based program transformations. *Inf. Comput.* 249 (2016), 160–189. doi:10.1016/J.IC.2016.04.003

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. doi:10.1145/3618003

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP (Lecture Notes in Computer Science, Vol. 2072)*. Springer, 2–27. doi:10.1007/3-540-45337-7_2

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023a. Graph IRs for Impure Higher-Order Languages – Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 236:1–236:30. doi:10.1145/3622813

Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023b. Graph IRs for Impure Higher-Order Languages (Supplement). arXiv:2309.08118 [cs.PL]

B Bunting and AS Murawski. 2025. Reachability types, traces and full abstraction. *40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2025)* (2025). https://ora.ox.ac.uk/objects/uuid:7ec96a90-1bb8-408d-90a8-26126a3bbc05

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *ECOOP (LIPIcs, Vol. 56)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:26. doi:10.4230/LIPICS.ECOOP.2016.5

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. doi:10.1007/978-3-642-36946-9_3

David Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64.

Sylvan Clebsch. 2017. *'Pony': co-designing a type system and a runtime.* Ph. D. Dissertation. Imperial College London, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.769552

Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. 2015a. Ownership and reference counting based garbage collection in the actor world. In *ICOOOLPS'2015*. ACM.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015b. Deny capabilities for safe, fast actors. In *AGERE SPLASH*. ACM, 1–12. doi:10.1145/2824815.2824816

Haotian Deng, Siyuan He, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2025. Complete the Cycle: Reachability Types with Expressive Cyclic References. *Proc. ACM Program. Lang.* 9, OOPSLA (2025). doi:10.1145/3763172

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 363–377. doi:10.1007/978-3-642-00590-9_26

Sophia Drossopoulou, Julian Mackay, Susan Eisenbach, and James Noble. 2025. Reasoning about External Calls. arXiv:2506.06544 [cs.PL]

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *POPL*. ACM, 270–282. doi:10.1145/1111037.1111062

Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca. 2019a. Tracing sharing in an imperative pure calculus. *Sci. Comput. Program.* 172 (2019), 180–202. doi:10.1016/J.SCICO.2018.11.007

Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019b. Flexible recovery of uniqueness and immutability. *Theor. Comput. Sci.* 764 (2019), 145–172. doi:10.1016/J.TCS.2018.09.001

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*. ACM, 21–40. doi:10.1145/2384616.2384619

Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *European Conference on Object-Oriented Programming*. Springer, 354–378. doi:10.1007/978-3-642-14107-2_17

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*. ACM, 271–285. doi:10.1145/117954.117975

Songlin Jia, Guannan Wei, Siyuan He, Yueyang Tang, Yuyan Bao, and Tiark Rompf. 2024. Escape with Your Self: Expressive Reachability Types with Sound and Decidable Bidirectional Type Checking. arXiv:2404.08217 [cs.PL]

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. doi:10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269. doi: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. doi:10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. doi:10.1145/2676726.2676980

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 696–723. doi:10.1007/978-3-662-54434-1_26

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. doi:10.1093/COMJNL/6.4.308

Danielle Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1040–1070. doi:10.1145/3649848

Nicholas D. Matsakis and Felix S. II Klock. 2014. The Rust language. In *HILT*. ACM, 103–104. doi:10.1145/2663171.2663188

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *POPL*. ACM, 557–570. doi:10.1145/2103656.2103722

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP (Lecture Notes in Computer Science, Vol. 1445)*. Springer, 158–185. doi:10.1007/BFb0054091

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19. doi:10.1007/3-540-44802-0_1

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 110:1–110:30. doi:10.1145/3341714

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages.* The MIT Press.

G. D. Plotkin. 1973. *Lambda Definability and Logical Relations.* Memorandum SAI-RM-4. University of Edinburgh. https://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf

Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. 2006. Generic ownership for generic Java. In *OOPSLA*. ACM, 311–324. doi:10.1145/1167473.1167500

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

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. ACM, 624–641. doi:10.1145/2983990.2984008

Jeremy Siek. 2013. Type safety in three easy lemmas. http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite step-indexing for termination. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434294

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. doi:10.2307/2271658

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

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

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. doi:10.1145/3158152

Fei Wang and Tiark Rompf. 2017. Towards Strong Normalization for Dependent Object Types (DOT). In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:25. doi:10.4230/LIPIcs.ECOOP.2017.27

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 393–424. doi:10.1145/3632856

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/INCO.1994.1093

Yichen Xu and Martin Odersky. 2023. Degrees of Separation: A Flexible Type System for Data Race Prevention. doi:10.48550/ARXIV.2308.07474

Received 2025-03-25; accepted 2025-08-12