Type, Ability, and Effect Systems: Perspectives on Purity, Semantics, and Expressiveness

YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

Programming benefits from a clear separation between pure, mathematical computation and impure, effectful interaction with the world. Existing approaches to enforce this separation include monads, type-and-effect systems, and capability systems. All share a tension between precision and usability, and each one has non-obvious strengths and weaknesses.

This paper aims to raise the bar in assessing such systems. First, we propose a semantic definition of purity, inspired by contextual equivalence, as a baseline independent of any specific typing discipline. Second, we propose that expressiveness should be measured by the degree of completeness, i.e., how many semantically pure terms can be typed as pure. Using this measure, we focus on minimal meaningful effect and capability systems and show that they are incomparable, i.e., neither subsumes the other in terms of expressiveness.

Based on this result, we propose a synthesis and show that *type, ability, and effect systems* combine their respective strengths while avoiding their weaknesses. As part of our formal model, we provide a logical relation to facilitate proofs of purity and other properties for a variety of effect typing disciplines.

1 Introduction

Computer programs bridge two worlds: the "pure" world of mathematical computation and the "impure" world of stateful physical reality. Some programs are dominantly computational in nature, with the goal of producing a discrete result upon termination. The nature of other programs is dominantly interactive, with the goal of observing and affecting state changes in the real world, be it to control machinery or to interact with users.[1] Many programs combine both aspects, and ultimately, all programs [2] are executed on stateful physical hardware. This creates tensions.

Maximizing Mathematical Reasoning From the perspective of computation, we often wish to view program terms as mathematical formulas. The essence of a term behaving in a "mathematical" way is that we can reason about it equationally rather than operationally, using intuitive techniques from middle-school algebra. These rest on the principles of extensionality and compositionality: it should be possible to identify a term with the result it computes, treating terms with indistinguishable results as equivalent, and to replace a subterm with an equivalent one without changing the overall result. This form of reasoning crucially depends on a term having a single, fixed result, which does not change depending on when or how often the term is evaluated. In other words, we want to view a program term as a mathematical function of its free variables, consistently producing equivalent outputs on equivalent inputs. We can only meaningfully say that $2 + 3 = 5$ because $2 + x$ depends only on the value of x but does not depend, *e.g.*, on the phase of the moon.

Stateful Physical Reality But in the real world, programs often behave "unmathematically", for two different reasons. The first reason is intentional: often the very purpose of a program is control, more than computation, *e.g.*, to drive machinery based on sensor inputs, or to perform tasks based on user interaction. Observing or affecting a state change in the physical world is the primary purpose. The second reason is accidental: even purely computational parts of a program are ultimately executed as a stream of machine instructions on a stateful microprocessor, which

[1] While the term "microcomputer" has fallen out of fashion, the term "microcontroller" remains in active use.
[2] Not all programs are "run": some are only written to illustrate an idea, or to serve as a Curry-Howard proof witness.

Authors' addresses: Yuyan Bao, Augusta University, Augusta, GA, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, West Lafayette, IN, USA, tiark@purdue.edu.

presents the abstraction of "bits" changing states in "memory" on top of some electrical circuitry. The tower of abstractions on top of the physical reality is colossal, spanning not only superscalar processors with out-of-order execution, but also virtual machines and optimizing compilers, various storage technologies, not to mention unreliable network connections to data centers on the other side of the planet.

It takes great care to orchestrate this stack in a way that preserves any mathematical illusion. At any level of granularity, a component may present a pure external interface based on an internal stateful implementation. Thus, what is pure or impure, and external or internal, always depends on *which aspects one is able to observe.*

Monads, Effects, Capabilities Given the desire to maximize mathematical reasoning and the potential consequences of accidental impurity where purity is assumed, programming languages have long sought to enforce purity in parts of a program, and to isolate pure from impure parts.

At the far end of the spectrum, proof assistants such as Coq, Agda, or Lean enforce total purity (including termination for logical consistency) but sacrifice general-purpose programming. At the other end, languages like C embrace impurity and provide little support for reasoning. Between these extremes, Monads [Moggi 1991; Wadler 1992], as in Haskell, can either encode effects using pure code or isolate them at the boundary with the runtime (the IO Monad). Effect type systems [Gifford and Lucassen 1986; Talpin and Jouvelot 1994] instead enrich types with annotations that track the occurrence of impure operations. A more recent proposal is capability-based control of effects, though based on earlier ideas [Dennis and Horn 1966; Miller 2006], where a type system tracks the flow of values that grant access to impure operations, without tracking the occurrence of impure operations directly [Boruch-Gruszecki et al. 2023; Brachthäuser et al. 2020; Craig et al. 2018; Gordon 2020; Osvald et al. 2016].

Challenges and Limitations Like with any form of typing discipline, all these approaches suffer from a tension between precision and usability. Monadic encodings add an indirection compared to direct-style code that can be awkward to deal with. Thus, for pragmatic reasons, even Haskell supports nontermination and exceptions in the "pure" fragment of the language (a Haskell function of type $\mathbb{N} \to \mathbb{N}$ is not guaranteed to be a mathematical function on the natural numbers). And while there is an abundance of sophisticated effect systems in the literature, very few are used outside of experimental settings. Those that do exist in widely used languages, such as checked exceptions in Java, are regarded by many as a failed experiment because the amount of type parameters required seems to outweigh the benefits of static checking [Odersky et al. 2021]. Capabilities have shown great promise for enabling effect-polymorphic behavior [Lucassen and Gifford 1988] without explicit parameterization, so there are high hopes that they may become a viable solution for modeling effects in mainstream languages [Boruch-Gruszecki et al. 2023; Brachthäuser et al. 2020; Osvald et al. 2016], but there are also trade-offs to consider with respect to effect systems, specifically when it comes to distinguishing actually *using* a capability to cause an effect from just *mentioning* a capability [Gordon 2020], *e.g.*, to pass it somewhere else.

1.1 Contributions

How should we compare and evaluate such diverse typing disciplines? While there are well-known results connecting monads and effects [Filinski 2010; Wadler and Thiemann 2003], less is known about the relationships between different effect and capability systems. The key goal of this paper is to raise the level of rigor in this area and establish a framework for comparing such diverse typing disciplines. To achieve this, we must first establish a common semantic foundation. Rather than relying solely on syntactic criteria or language-specific features, we advocate for a semantic approach based on a contextual notion of purity, inspired by contextual equivalence.

Using this framework, we can then compare the expressiveness of different typing disciplines by measuring their degree of completeness in characterizing semantically pure terms. This allows us to assess how well each system captures the intended notion of purity and to identify their respective strengths and weaknesses. We illustrate this approach by formalizing and comparing canonical binary effect and capability systems, demonstrating their incomparability in terms of expressiveness. To address the identified weaknesses, we propose a synthesis of effect and capability systems that combines their respective strengths. This hybrid approach aims to leverage the advantages of both paradigms while mitigating their individual limitations.

To facilitate further research and practical reasoning about purity, we present a logical relation that enables formal proofs of purity across different typing disciplines. This relation is designed to be adaptable, supporting a variety of language features and type system designs.

In summary, the main contributions of this paper are:

- A framework for measuring the expressiveness of effect typing disciplines based on their ability to characterize semantically pure terms as pure (Section 2).
- A semantic definition of purity, independent of any particular type system, inspired by contextual equivalence (Section 3).
- Formalization and comparison of canonical binary effect and capability systems, demonstrating their incomparability with respect to expressiveness (Sections 4 and 5).
- A synthesis of effect and capability systems, combining their respective strengths to address identified weaknesses (Section 6).
- A general logical relation suitable for proving purity, applicable to a range of typing disciplines and language designs (Section 7).

We discuss related work in Section 8, and offer concluding remarks in Section 9. Mechanized proofs and accompanying developments are available online.[3]

2 Semantics of Effects and Purity

We begin by establishing some intuition behind our framework for reasoning about effects and purity in programming languages. This framework will allow us to compare different effect typing disciplines and their expressiveness.

2.1 Expressiveness of Effect Typing Disciplines

While the term "expressiveness" is still frequently used informally, Felleisen's notion of macro-expressibility [Felleisen 1991] provides a formal foundation for understanding the expressive power of programming languages. However, when it comes to type systems, the meaning of expressiveness is less well established.

One reasonable and, we believe, common interpretation is that a more expressive type system can type more programs, such as in the case of System F versus simply-typed λ-calculus [Tobin-Hochstadt 2011]. This of course assumes that we consider only type systems that are sound, as unsound type systems can trivially type all programs. Thus, expressiveness can be understood as the degree of completeness of a type system, *i.e.*, the proportion of all semantically well-typed programs (in the sense of "well-typed programs don't go wrong') that are well-typed syntactically.

But can we apply the same measure to effect typing disciplines? If we only consider the number of typable terms, the metric becomes trivial: since effects refine an existing type system, we could simply type all terms as impure. Clearly, this does not capture the essence of expressiveness for effect systems. The key realization, of course, is that for an effect systems to be sound, it should reflect stronger semantic properties than just "well-typed programs don't go wrong". These semantic

[3] https://github.com/tiarkrompf/reachability

(Abs:Any)	$\lambda x.\ t$	Function value	Pure
(Diverge)	$(\lambda x.x\ x)\ (\lambda x.x\ x)$	Nontermination	Impure
(Alloc)	`ref` t	Resource allocation	Impure
(Use:Read)	`!`a	Resource use	Impure
(Use:Write)	a `:=` t	Resource use	Impure
(Mention)	`let` $x = a$ `in` t	Resource mention	Pure iff t pure
(Mask:Read)	`let` $x =$ `ref` t `in !`x	Masked resource use	Pure iff t pure
(Mask:Write)	`let` $x =$ `ref` t `in` x `:=` t'	Masked resource use	Pure iff t, t' pure

Fig. 1. Examples illustrating intuitively pure and impure terms (a refers to a mutable reference).

properties depend on the specific effect system so they are hard to capture with any generality. More disturbingly, they are often not precisely defined, especially when effects are modeled purely syntactically: unlike with simple type systems, there is no notion of effects being preserved under reduction from terms to values, and there is no canonical equivalent to "getting stuck" for effect typing violations. Any attempt at artificially forcing stuckness in a reduction semantics is prone to overlooking some possible error cases, so we believe that a more semantic approach is indispensable for a rigorous treatment.

Irrespective of the precise semantic effect soundness properties of any specific effect system, we propose that any meaningful effect system should at least satisfy the property that syntactically pure terms are also semantically pure. Thus, a natural way to measure the expressiveness of different effect systems is by asking: how many semantically pure terms can the system type as pure? This leads to the next question: what exactly does it mean for a term to be semantically pure?

2.2 Intuitive Definitions of Purity

Perhaps surprisingly, there does not seem to be a universally agreed-upon definition of what it means for a term to be pure. However, there are several common intuitions that we can build upon. We propose that the following three intuitions are equivalent and capture the essence of purity:

(1) **Pure terms are those that support mathematical reasoning.** Perhaps more precisely, *algebraic* reasoning and manipulation.
(2) **Pure terms are mathematical functions of their free variables.** This stresses the principles of extensionality and compositionality.
(3) **Pure terms terminate, have deterministic results, and evaluate without effects.** Effects include (1) primary effects, *i.e.*, external state changes such as IO; (2) observable side effects, *i.e.*, internal state changes that impact the result of evaluating any *other term*.

Among the possible effects of a term, externally observable effects, *e.g.*, IO (file/network read and write), are the easiest to capture with a type system, hence we will mostly disregard them. The more interesting case is that of observable side effects, *i.e.*, internal state changes that impact the result of evaluating other terms. Figure 1 shows examples of pure and impure terms, including cases of effect polymorphism, effect masking, and use vs. mention situations.

2.3 Pure Functions, Pure Ability

To state whether a function application is pure, we also need the concept of a *pure function*. We provide the following intuitive definition, based on a function's extensional behavior:

Definition 2.1 (Pure Function – Informal). A term f is a pure function iff, when f is applied to an argument t, then the overall application term $f\ t$ is pure, modulo any impurity caused by evaluating either f or t directly.

(Abs:Id)	$\lambda x.\ x$	Pure function	Pure ability
(Abs:Diverge)	$\lambda x.\ \omega$	Impure function	Impure ability
(Abs:Alloc)	$\lambda x.\ \texttt{ref}\ x$	Impure function	Impure ability
(Abs:Leak)	$\lambda x.\ a$	Pure function	Impure ability
(Abs:Use)	$\lambda x.\ !a$	Impure function	Impure ability
(Abs:UseArg)	$\lambda x.\ !x$	Impure function	Pure ability
(Abs:Mention)	$\lambda x.\ \texttt{let}\ x = a\ \texttt{in true}$	Pure function	Pure ability
(Abs:Capture)	$\lambda x.\ \lambda y.\ !a$	Pure function	Impure ability
(Abs:Poly)	$\lambda f.\ \lambda x.\ f\ x$	Impure function	Pure ability

Fig. 2. Examples illustrating intuitively pure and impure functions (which are all pure terms here, but could be the result of an impure computation), and the concept of *ability* or *potential* of a value: namely, to be the source of impure behavior when put in a specific context. Values with impure ability are what we intuitively understand as *capabilities*.

Figure 2 shows examples of pure and impure functions. To think about effects in terms of using capabilities, we introduce the notion of a term's *ability* (following terminology introduced by Lindley et al. [2017]). An impure function may still have pure ability in the sense that it can't perform an effect on its own, but it needs another object, a *capability*, to cause this effect.

Definition 2.2 (Pure-Ability Function – Informal). A term f is a pure-ability function iff, when f is applied to a pure-ability argument t, then the overall application term $f\ t$ is pure (modulo impurity caused by evaluating f or t), and the result is also a pure-ability value. In addition to pure-ability functions, primitives also count as pure-ability values.

Intensionally, ability describes whether a term's result has captured any effect-inducing capabilities, such as mutable references. Extensionally, ability describes whether a term is part of the pure fragment of a language or not. For example, $\lambda x.\ !x$ is not a pure function because it dereferences its argument x, but we say it has pure ability because the reference needs to be provided externally, it has not been captured by the function. On the other hand, $\lambda f.\ \lambda x.\ f\ x$ is not a pure function because when applied to the argument $\lambda x.\ !a$, a pure term, the application is impure. Likewise, a pure function may also have impure ability: $\lambda x.\ a$ is a pure function, but it returns a reference it has captured. We thus say it has impure ability. Finally, $\lambda x.\ \texttt{ref}\ x$ is an impure function, and it also has impure ability. It does not capture the ref it returns, but it still returns a value capable of inducing an effect elsewhere.

3 A Semantic Definition of Purity

In the following, we propose a semantic notion of purity that matches the intuition outlined above. The definition is so simple and straightforward that we are almost certainly not the first to operate with this definition in mind, at least informally. However, we have not found it spelled out explicitly in the literature.

3.1 Formalizing A Notion of Observational Purity

3.1.1 Syntax and Semantics To ground this discussion, we fix some minimal requirements on the program syntax and semantics. We consider the syntax consisting of terms t, values v, and states σ, along with value environments H, which map variables to values.

b	:=	$\texttt{true} \mid \texttt{false} \mid \ldots$	Constants
t	:=	$b \mid x \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \ldots$	Terms
v	:=	$b \mid \ldots$	Values
σ	:=	$\ldots$	State
H	:=	$\varnothing \mid H; (x, v)$	Value Environments

We intentionally leave the precise structure of terms, values, and state underspecified at this stage. However, we do assume that the term language supports sequencing via let-bindings. Note here the connection with Moggi's computational λ-calculus [Moggi 1991], which introduced monads as a model of effectful computation. Among the set of values, we assume a distinguished subset of atomic primitives (*e.g.*, integers and booleans), which serve as effect-free observables.

The operational semantics defines evaluation as a relation of the form:

$$H,\ \sigma,\ t \Downarrow \sigma',\ v$$

This means that the evaluation of term t with value environment H and pre-state σ terminates with result value v and post-state σ'. We assume that the evaluation relation itself is deterministic, i.e., there exists no more than a single v and σ' for any given t, H, and σ. If evaluation is undefined in the case of an error or divergence, and thus no v and σ' exists for a given t, H, and σ, we also write:

$$H,\ \sigma,\ t \Uparrow$$

Constants evaluate to themselves, and the evaluation of `let` expressions is strict. This is important to guarantee proper sequencing. Later, we will formally extend the language with λ-abstractions and mutable references (see Section 3.2). Extensions with IO, exceptions, etc., are also possible, but we prefer to keep our core definitions independent of any such specific features.

In the following, we will largely keep the environment H and state σ implicit. So when clear from the context, we will just write $t \Uparrow$ and $t \Downarrow v$ to mean that t evaluates in its associated H and σ.

3.1.2 Observational Equivalence Our notion of *observational* (or *behavioral*) purity is inspired crucially by the notion of observational (or behavioral) *equivalence*, typically attributed to Morris' 1969 PhD thesis (there called *extensional equivalence*) [Morris 1969]. This turned out to be a key milestone in the ability to reason about programs equationally, as the straightforward identification of terms that evaluate to *identical* values is too restrictive. One could consider equivalence of result values modulo some conversion rules, such as those defined in Morris' thesis [Morris 1969]. However, in a programming language with references, two allocation terms, *e.g.*, `ref 0` and `ref 0` can never be reduced to the same store location due to the nondeterminism of store allocation in most semantic models.

Instead, one sets up a weaker form of auxiliary equivalence first that only considers termination behavior and atomic result values such as integers and Booleans[4] (for all the definitions below, we assume that evaluation takes place in two equivalent pre-states σ_1 and σ_2, for a suitable notion of state equivalence).

Definition 3.1 (Operational Equivalence). Two terms t_1 and t_2 are operationally equivalent $t_1 \simeq t_2$ iff both have the same termination behavior and they agree on any atomic result values b:

$$t_1 \Uparrow \ \Leftrightarrow\ t_2 \Uparrow \qquad \wedge \qquad t_1 \Downarrow b \ \Leftrightarrow\ t_2 \Downarrow b$$

Then, the key idea is to treat all those terms as equivalent for which the *program itself* cannot observe any difference. So, no possible context should exist that would expose any difference in behavior:

Definition 3.2 (Observational Equivalence). Two terms t_1 and t_2 are observationally equivalent $t_1 \cong t_2$ iff for all contexts C:

$$C[t_1] \simeq C[t_2]$$

[4]There are different names for this auxiliary equivalence in the literature. Often the requirement to agree on atomic result values is dropped (*i.e.*, requiring just the same termination behavor) but we prefer to include this condition, as it is necessary for terminating languages.

Syntax $\boxed{t ::= \ldots}$

b	$:=$	$\texttt{true} \mid \texttt{false}$	Constants
t	$::=$	$b \mid x \mid \lambda x.\, t \mid t_1\ t_2 \mid \texttt{ref}\ t \mid\ !t \mid t_1 := t_2 \mid t_1 \otimes t_2$	Terms
$\otimes$	$::=$	$\vee \mid \wedge$	Boolean Operator
v	$::=$	$b \mid \ell \mid \langle H,\ \lambda x.\, t \rangle$	Values
H	$::=$	$\varnothing \mid H; (x, v)$	Value Environment
σ	$:=$	$\varnothing \mid \sigma; (\ell, v)$	Store

Operational Semantics $\boxed{H,\ \sigma,\ t \Downarrow \sigma',\ v}$

$$H,\ \sigma,\ b \Downarrow \sigma,\ b \quad \text{(E-CST)} \qquad \frac{H(x) = v}{H,\ \sigma,\ x \Downarrow \sigma,\ v} \quad \text{(E-VAR)}$$

$$\frac{H,\ \sigma,\ t_1 \Downarrow \sigma',\ b_1 \quad H,\ \sigma',\ t_2 \Downarrow \sigma'',\ b_2}{H,\ \sigma,\ t_1 \otimes t_2 \Downarrow \sigma'',\ b_1 [\![\otimes]\!] b_2} \quad \text{(E-BIN)} \qquad H,\ \sigma,\ \lambda x.\, t \Downarrow \sigma,\ \langle H,\ \lambda x.\, t \rangle \quad \text{(E-ABS)}$$

$$\frac{H,\ \sigma,\ t \Downarrow \sigma',\ v \quad \ell \notin \mathrm{dom}(\sigma')}{H,\ \sigma,\ \texttt{ref}\ t \Downarrow \sigma'; (\ell, v),\ \ell} \quad \text{(E-REF)} \qquad \frac{H,\ \sigma,\ t_1 \Downarrow \sigma',\ \langle H',\ \lambda x.\, t \rangle \quad H,\ \sigma',\ t_2 \Downarrow \sigma'',\ v \quad H'; (x, v),\ \sigma'',\ t \Downarrow \sigma''',\ v}{H,\ \sigma,\ t_1\ t_2 \Downarrow \sigma''',\ v} \quad \text{(E-APP)}$$

$$\frac{H,\ \sigma,\ t \Downarrow \sigma',\ \ell \quad \sigma'(\ell) = v}{H,\ \sigma,\ !t \Downarrow \sigma',\ v} \quad \text{(E-GET)} \qquad \frac{H,\ \sigma,\ t_1 \Downarrow \sigma',\ \ell \quad H,\ \sigma',\ t_2 \Downarrow \sigma'',\ v}{H,\ \sigma,\ t_1 := t_2 \Downarrow \sigma''[\ell \mapsto v],\ \texttt{true}} \quad \text{(E-PUT)}$$

Fig. 3. Syntax and Semantics.

3.1.3 Observational Purity In analogy, we want to express that pure expressions are those that do nothing else (observably) than computing their results. So, no possible context should exist that would expose any difference in behavior between the term itself, possible evaluated an arbitrary numbers of times (including zero) by the context, and evaluating the term once outside the context and then reusing its result bound to a variable (with strict binding semantics):

Definition 3.3 (Observational Purity). A term t is observationally pure iff for all contexts C and fresh variables x:

$$\texttt{let}\ x = t\ \texttt{in}\ C[x] \simeq C[t]$$

In a language that includes the call-by-value λ calculus, the left-hand side $\texttt{let}\ x = t\ \texttt{in}\ C[x]$ can be encoded as λ application $(\lambda x.C[x])\ t$. This suggests a nice symmetric interpretation of pure terms as (a) invariant under η-expansion of any context C into a λ abstraction, and (b) invariant under β-reduction in any application position. The latter interpretation is consistent with Sabry's definition of purity [Sabry 1998], modulo our additional requirement for termination.

In terms of terminology, we consider a term observationally impure if it is not observationally pure. When clear from the context, we will drop the "observationally".

3.2 Discussion

We now fix a concrete language, with which we examine our definition of purity with a few examples. Fig. 3 defines the language syntax and semantics, which extends STLC with mutable references and Boolean expressions. The definition is standard, and can be found in the textbook [Pierce 2002]. We use a big step semantics with a value environment H and a store σ, where σ is a partial function that maps from locations to values. As before, we write $H,\ \sigma,\ t \Downarrow \sigma',\ v$ to mean that term t evaluates to value v, resulting in a store transition from σ to σ'. In the semantics, we write $[\![\otimes]\!]$ to

denote the interpretation of a syntactic Boolean operator as its corresponding meta-level operator. Other definitions of the semantics are standard.

Following [Amin and Rompf 2017; Ernst et al. 2006; Siek 2013; Wang and Rompf 2017], we extend the big-step semantics $\Downarrow$ to a total evaluation function by adding a numeric fuel value and explicit timeout and Error results. This semantic definition leads to strong soundness results in Section 7.

Nontermination A nonterminating expression is *impure*, as expected. Consider an example: $\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$. The constant context $C[.] = \texttt{true}$ distinguishes between the term and its result bound to a variable: $\texttt{let}\ x = \omega\ \texttt{in true}$ diverges, but $C[\omega] = \texttt{true}$ does not. With a level of abstraction, however, we obtain a *pure* expression: $\lambda x.\ \omega$ behaves the same in all contexts for both $\texttt{let}\ x = \lambda x.\ \omega\ \texttt{in}\ C[x]$ and $C[\lambda x.\ \omega]$. However, this expression is not a *pure function*, because there are impure applications, and in fact any application will be impure (e.g., $\omega\ \texttt{true}$). Thus, it is also not a *pure-ability function*.

Resource Allocation An allocation is *impure*. Consider the expression $\texttt{let}\ x = \texttt{ref}\ t\ \texttt{in}\ C[x]$. This means that only a single location is allocated and then reused through the variable x; whereas $C[\texttt{ref}\ t]$ means that potentially many locations (or none) are allocated. It is easy to craft a context to observe this difference. For example, $\texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in}\ x := !x + 1$; $!x$ yields value 1, reflecting the single reference being updated. In contrast, $(\texttt{ref}\ 0) := !(\texttt{ref}\ 0) + 1; !(\texttt{ref}\ 0)$ repeatedly allocates fresh locations and thus yields value 0.

Resource Mentioning and Usage Merely referencing a variable bound in the environment is pure, but dereferencing it is not. Consider the expression $\texttt{let}\ x = a\ \texttt{in}\ C[x]$, where a is a mutable reference in the environment. Since both x and a are variables, $\texttt{let}\ x = a\ \texttt{in}\ C[x]$ and $C[a]$ must behave the same. However, if we consider $\texttt{let}\ x = !a\ \texttt{in}\ C[x]$, then the two expressions may behave differently. Thus, we achieve a proper distinction between *mentioning* a resource and *using* it. This distinction also extends to from pure terms to pure functions.

Effect Masking The expression $\texttt{let}\ x = \texttt{ref}\ t\ \texttt{in}\ !x$ is *pure*, despite the internal use of a mutable reference. It remains pure if we insert a "proper" side effect, such as a write operation on x, *e.g.*, $\texttt{let}\ x = \texttt{ref}\ t\ \texttt{in}\ x := !x$. This demonstrates the strength of the notion of observational purity: if store changes or other side effects cannot be observed, it is as if they did not exist.

4 Effect Type Systems (λ_e)

We now consider our semantic notion of purity in the context of type systems that distinguish pure and impure terms. The classic approach to do so is via a type and effect system. As basis, we consider a version of simply-typed λ calculus with first-order mutable references, restricted to hold Boolean values.

In Fig. 4, we recap a canonical binary effect system, distinguishing pure ($\bot$) and impure ($\top$) expressions. The effect ordering is defined as $\bot <: \top$. We use the operator $\triangleright$ for flow-insensitive sequential effect composition, *i.e.*, $e_1 \triangleright e_2$ computes the least upper bound of the effects e_1 and e_2.

4.1 Formalization

The effect typing rules are shown in Fig. 4. The effect typing judgment is in the form of $\Gamma^e \vdash_e t : T^e\ e$, where e is a binary effect. It means that under the assumption Γ^e, the term t has type T^e and its evaluation may induce the observable effect e. Function types carry an effect annotation e for its latent effect.

Rules for mutable references (T-Ref-E, T-Get-E and T-Put-E) are impure expressions, each assigned the effect $\top$. Constant terms (T-Cst-E), variables (T-Var-E) and function abstractions (T-Abs-E) are pure, each assigned the effect $\bot$. For compound terms (*i.e.*, T-App-E and T-Bin-E),

Type & Effects $\boxed{T^e ::= \ldots}$ $\boxed{e ::= \ldots}$

$$T^e ::= \texttt{Bool} \mid \texttt{Ref} \mid T_1^e \xrightarrow{e_2} T_2^e \qquad e ::= \bot \mid \top \qquad e \rhd e' := e \vee e'$$

Term Typing $\boxed{\Gamma^e \vdash_e t : T^e\ e}$

$$\Gamma^e \vdash_e b : \texttt{Bool}\ \bot \quad \text{(T-Cst-E)} \qquad \frac{\Gamma^e(x) = T^e}{\Gamma^e \vdash_e x : T^e\ \bot} \quad \text{(T-Var-E)}$$

$$\frac{\Gamma^e \vdash_e t_1 : \texttt{Bool}\ e_1 \quad \Gamma^e \vdash_e t_2 : \texttt{Bool}\ e_2}{\Gamma^e \vdash_e t_1 \otimes t_2 : \texttt{Bool}\ e_1 \rhd e_2} \quad \text{(T-Bin-E)} \qquad \frac{\Gamma^e, x : T_1^e \vdash_e t : T_2^e\ e_2}{\Gamma^e \vdash_e \lambda x.\, t : (T_1^e \xrightarrow{e_2} T_2^e)\ \bot} \quad \text{(T-Abs-E)}$$

$$\frac{\Gamma^e \vdash_e t : \texttt{Bool}\ e}{\Gamma^e \vdash_e \texttt{ref}\ t : \texttt{Ref}\ \top} \quad \text{(T-Ref-E)}$$

$$\frac{\Gamma^e \vdash_e t : \texttt{Ref}\ e}{\Gamma^e \vdash_e\ !t : \texttt{Bool}\ \top} \quad \text{(T-Get-E)} \qquad \frac{\Gamma^e \vdash_e t_1 : (T_1^e \xrightarrow{e_2} T_2^e)\ e_f \quad \Gamma^e \vdash_e t_2 : T_1^e\ e_1}{\Gamma^e \vdash_e t_1\ t_2 : T_2^e\ e_f \rhd e_1 \rhd e_2} \quad \text{(T-App-E)}$$

$$\frac{\Gamma^e \vdash_e t_1 : \texttt{Ref}\ e_1 \quad \Gamma^e \vdash_e t_2 : \texttt{Bool}\ e_2}{\Gamma^e \vdash_e t_1 := t_2 : \texttt{Bool}\ \top} \quad \text{(T-Put-E)} \qquad \frac{\Gamma^e \vdash_e t : T_1^e\ e_1 \quad T_1^e <: T_2^e \quad e_1 <: e_2}{\Gamma^e \vdash_e t : T_2^e\ e_2} \quad \text{(T-Sub-E)}$$

Subtyping $\boxed{T_1^e <: T_2^e}$ $\boxed{e_1 <: e_2}$

$$\texttt{Bool} <: \texttt{Bool} \quad \text{(S-Cst-E)} \qquad \frac{T_3^e <: T_1^e \quad T_2^e <: T_4^e \quad e_1 <: e_2}{T_1^e \xrightarrow{e_1} T_2^e <: T_3^e \xrightarrow{e_2} T_4^e} \quad \text{(S-Abs-E)}$$

$$\texttt{Ref} <: \texttt{Ref} \quad \text{(S-Tref-E)}$$

$$\bot <: \top \quad \text{(S-E)} \qquad \bot <: \bot \quad \text{(S-}\bot\text{-E)} \qquad \top <: \top \quad \text{(S-}\top\text{-E)}$$

Fig. 4. Effect Type System λ_e.

Free Variables $\boxed{\mathrm{fv}(t)}$

$$\begin{array}{llll} \mathrm{fv}(b) = \emptyset & & \mathrm{fv}(x) = \{x\} & \\ \mathrm{fv}(\lambda x.\, t) = \mathrm{fv}(t) \setminus \{x\} & & \mathrm{fv}(t_1\ t_2) = \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2) & \\ \mathrm{fv}(\texttt{ref}\ t) = \mathrm{fv}(t) & & \mathrm{fv}(!t) = \mathrm{fv}(t) & \\ \mathrm{fv}(t_1 := t_2) = \mathrm{fv}(t) \cup \mathrm{fv}(t_2) & & \mathrm{fv}(t_1 \otimes t_2) = \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2) & \end{array}$$

Fig. 5. Computing free variables of a given term (standard).

the final effect is computed by composing the effects of their sub-terms. Effect composition $e_1 \rhd e_2$ computes the least upper bound of e_1 and e_2.

The subtyping rules are standard. In particular, the rule for function types (S-Abs-E) demands contravariance in the argument type and covariance in the result type.

4.2 Metatheory

The core metatheoretic development is fairly standard. In this work, we define program equivalence in terms of contextual equivalence, and establish it via binary logical relations (details in Section 7).

In particular, we say terms t_1 and t_2 are contextually equivalent, written as $\Gamma^e \models_e t_1 \cong_e t_2 : T^e\ e$, if they have the same observable behavior when placed in any program context C. To achieve that,

we use the semantic typing judgment, which is defined as $\Gamma^e \models_e t_1 \approx_{\log_e} t_2 : T^e\ e \stackrel{\text{def}}{=} \forall(\hat{H}, \mathtt{W}) \in G[\![\Gamma^e\ \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)]\!], (\hat{H}, \mathtt{W}, t_1, t_2) \in \mathcal{E}[\![T^e\ e]\!]$, where $\hat{H}$ is a pair of related value environments, $\mathtt{W}$ is a world, $G[\![\Gamma^e\ \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)]\!]$ is the logical interpretation of typing environment, and $\mathcal{E}[\![T^e\ e]\!]$ is the value interpretation of terms, which entails semantic type soundness and termination. We define $\mathrm{fv}(t)$ to compute the free variables of term t, which is standard, shown in Fig. 5. The detailed definitions and results will be presented in Section 7. The main results are summarized below.

Theorem 4.1 (Fundamental Property). *If* $\Gamma^e \vdash_e t : T^e\ e$, *then* $\Gamma^e \models_e t \approx_{log_e} t : T^e\ e$.

The following theorem shows the soundness of our binary logical relations with respect to contextual equivalence.

Theorem 4.2 (Contextual Equivalence). *if* $\Gamma^e \vdash_e t_1 : T^e\ e$ *and* $\Gamma^e \vdash_e t_2 : T^e\ e$, *then* $\Gamma^e \models_e t_1 \approx_{log_e} t_2 : T^e\ e$ *implies* $\Gamma^e \models_e t_1 \cong_e t_2 : T^e\ e$.

In the following theorem, we write $C : (\Gamma^e; T^e\ e) \Rrightarrow (\Gamma^{e'}; T^{e'}\ e')$ to mean that the context C is a program of type $T^{e'}\ e'$ (closed under $\Gamma^{e'}$) with a hole that can be filled with any program of type $T^e\ e$ (closed under Γ^e). The typing rules for well-typed contexts can be found in Section 7.

Theorem 4.3 (Effect Safety of Effect Type System). *A syntactically pure expression is semantically (observationally) pure:*

$$\Gamma^e \vdash_e t : T^e\ \bot \quad \Rightarrow \quad \forall C : (\Gamma^e; T^e\ \bot) \Rrightarrow (\Gamma^{e'}; T^{e'}\ e).\ \Gamma^{e'} \models_e (\lambda x.\ C[x])\ t \cong_e C[t] : T^{e'}\ e$$

We demonstrate that pure terms have expected algebraic properties in mathematical formulas, *e.g.*, they allow reordering of operand expressions.

Theorem 4.4 (Reordering). *Pure terms can be evaluated in any order,* i.e., *swapping arguments of commutative operations is legal:*

$$\Gamma^e \vdash_e t_1 \otimes t_2 : \mathtt{Bool}\ \bot \quad \Rightarrow \quad \Gamma^e \models_e t_1 \otimes t_2 \approx_{log_e} t_2 \otimes t_1 : \mathtt{Bool}\ \bot$$

4.3 Discussion

Effects are Properties of Evaluating an Expression (Positive) Effects classify properties of computations rather than values. The system is terminating, so we cannot type ω. But with a slight extension to allow functions to call themselves recursively this becomes possible, and we can treat recursive calls as effectful. A more refined scheme could detect structurally recursive calls and treat them as pure. This, as one would expect, leads to ω being syntactically impure and $\lambda x.\ \omega$ syntactically pure.

To not complicate the presentation and formalism with multiple kinds of effects, we do not further consider nontermination in this paper – we rather focus on store effects, and the exact same considerations can be made with `ref true` instead of ω.

Possibility for Flow-Sensitive Reasoning (Positive) We can achieve flow-sensitive reasoning, *e.g.*, by considering an alternative system with a single global boolean state and two operations: `tic` and `toc`. Each operation expects the state value to be set one way and change it to the other, or fail if it has the wrong value. Thus, legal sequences are `tic`,`toc`,`tic`,... etc.

Such operational behavior can be modeled with a more specialized effect composition operation $e \triangleright e'$, which captures the legal sequences. That is, both `tic` $\triangleright$ `toc` and `toc` $\triangleright$ `tic` are well-defined, but invalid compositions, *e.g.*, `tic` $\triangleright$ `tic`, are undefined.

Allocations are Impure: Hence, they have Effects (Neutral, light negative) We discussed earlier that allocations are impure semantically, as they interact with the store, and yield nondeterministic store locations. In our binary effect system, impurity is expressed via the effect annotation $\top$. Thus, allocations are treated the same as read and write effects.

We could refine the effect hierarchy with a more fine-grained hierarchy, *i.e.*, distinguishing read, write and allocation effects. Fundamentally, the refinement still treats allocation as an effect, *i.e.*, as a property of evaluting an expression. However, from a semantic perspective, what we often want to characterize is not merely that an allocation occurred, but rather that *the resulting value is a fresh location*, *i.e.*, a property of the result value with respect to the context, instead. We will come back to this later.

Incompleteness, no Effect Masking (Negative) There are many terms that are semantically pure but cannot be assigned a syntactically pure type. By necessity, any approximation of nontermination cannot be precise. And we also do not obtain effect masking: since every allocation has to be effectful, an expression such as `let` x `=` `ref` t `in !`x cannot be assigned a pure type syntactically.

Of course, if we are willing to enrich the effect lattice with extra information, it becomes possible to model effect masking and other features. Benton et al.'s work [Benton et al. 2014], *e.g.*, achieves this by considering effects relative to store regions, and masking out effects on regions that are no longer observable, *i.e.*, are no longer referenced anywhere in the term or the environment.

No Effect Polymorphism (Negative) Our simple effect system shows its limitations in the presence of higher-order functions or objects in an object-oriented programming language, where every method is a higher-order function (parametric in the receiver). Consider a function such as `map(f)`, whose effect depends on the effect of the function argument `f`. Clearly, our binary effect system lacks the expressiveness to abstract over effects. One natural extension is to adopt a polymorphic effect system [Lucassen and Gifford 1988; Talpin and Jouvelot 1994], introducing abstract effect parameters to our system. However, such systems come at a much greater expense in complexity: essentially every higher-order function needs additional effect parameters to abstract over the effect of the function arguments, leading to significant annotation overhead or code duplication [Boruch-Gruszecki et al. 2023].

The problem becomes even more pronounced in the presence of deeply higher-order or dynamically dispatched code. A function's effect must include all the effects of all functions it may transitively invoke, preventing effect systems from practical adoption [Boruch-Gruszecki et al. 2023; Rytz 2014].

5 Ability Type Systems (λ_a)

As an alternative to effect type systems, systems that track the flow of capabilities have received renewed interest in recent times. Rather than treating effects as a property of how a term evaluates, these systems track the flow of *capabilities* or *resources* (such as mutable references).

This, most importantly, has the promise of dealing more gracefully with effect polymorphism: capabilities can be passed down the callgraph through the environment without polluting the types of higher-order functions with additional quantifiers.

While a number of concrete type systems with these goals in mind have been proposed, *e.g.*, the work of Xhebraj et al. [2022] and Osvald et al. [2016], Scala Capturing Types [Boruch-Gruszecki et al. 2023], and Reachability Types [Bao et al. 2021; Wei et al. 2024]. However, a fundamental study of their effect semantics has not been conducted.

In the following, we present and analyze a canonical binary ($a := \bot \mid \top$) "ability" type system from our perspective of observational purity.

Types & Ability $\boxed{T^a ::= \ldots}$ $\boxed{a ::= \ldots}$

$$T^a \quad := \quad \texttt{Bool} \mid \texttt{Ref} \mid T_1^a\ a_1 \to T_2^a\ a_2 \qquad a \quad := \quad \bot \mid \top$$

Term Typing $\boxed{\Gamma^a \vdash_a t : T^a\ a}$

$$\frac{}{\Gamma^a \vdash_a b : \texttt{Bool}\ \bot}\ \text{(T-Cst-A)} \qquad \frac{\Gamma^a(x) = T^a\ a}{\Gamma^a \vdash_a x : T^a\ a}\ \text{(T-Var-A)}$$

$$\frac{\Gamma^a \vdash_a t_1 : \texttt{Bool}\ a_1 \quad \Gamma^a \vdash_a t_2 : \texttt{Bool}\ a_2}{\Gamma^a \vdash_a t_1 \otimes t_2 : \texttt{Bool}\ \bot}\ \text{(T-Bin-A)} \qquad \frac{\Gamma^a, x : T_1^a\ a_1 \vdash_a t : T_2^a\ a_2 \quad \Gamma^a(\text{fv}(t) \setminus x) <: a_f}{\Gamma^a \vdash_a \lambda x.\ t : (T_1^a\ a_1 \to T_2^a\ a_2)\ a_f}\ \text{(T-Abs-A)}$$

$$\frac{\Gamma^a \vdash_a t : \texttt{Bool}\ a}{\Gamma^a \vdash_a \texttt{ref}\ t : \texttt{Ref}\ \top}\ \text{(T-Ref-A)}$$

$$\frac{\Gamma^a \vdash_a t : \texttt{Ref}\ a}{\Gamma^a \vdash_a\ !t : \texttt{Bool}\ \bot}\ \text{(T-Get-A)} \qquad \frac{\Gamma^a \vdash_a t_1 : (T_1^a\ a_1 \to T_2^a\ a_2)\ a_f \quad \Gamma^a \vdash_a t_2 : T_1^a\ a_1}{\Gamma^a \vdash_a t_1\ t_2 : T_2^a\ a_2}\ \text{(T-App-A)}$$

$$\frac{\Gamma^a \vdash_a t_1 : \texttt{Ref}\ a_1 \quad \Gamma^a \vdash_a t_2 : \texttt{Bool}\ a_2}{\Gamma^a \vdash_a t_1 := t_2 : \texttt{Bool}\ \bot}\ \text{(T-Put-A)} \qquad \frac{\Gamma^a \vdash_a t : T_1^a\ a_1 \quad T_1^a <: T_2^a \quad a_1 <: a_2}{\Gamma^a \vdash_a t : T_2^a\ a_2}\ \text{(T-Sub-A)}$$

Augmented Term Typing with Ambient Ability $\boxed{\Gamma^a\ a' \vdash_a t : T^a\ a}$

$$\frac{\Gamma^a \vdash_a t : T^a\ a \quad \Gamma^a(\text{fv}(t)) <: a'}{\Gamma^a\ a' \vdash_a t : T^a\ a}$$

Subtyping $\boxed{T_1^a <: T_2^a}$ $\boxed{a_1 <: a_2}$

$$\texttt{Bool} <: \texttt{Bool}\ \text{(S-Cst-A)} \qquad \texttt{Ref} <: \texttt{Ref}\ \text{(S-Tref-A)}$$

$$\frac{T_3^a <: T_1^a \quad T_2^a <: T_4^a \quad a_3 <: a_1 \quad a_2 <: a_4}{(T_1^a\ a_1 \to T_2^a\ a_2) <: (T_3^a\ a_3 \to T_4^a\ a_4)}\ \text{(S-Abs-A)}$$

$$\bot <: \top\ \text{(S-A)} \qquad \bot <: \bot\ \text{(S-}\bot\text{-A)} \qquad \top <: \top\ \text{(S-}\top\text{-A)}$$

Fig. 6. Ability Type System λ_a.

5.1 Formalization

The ability typing rules are shown in Fig. 6. The ability typing judgment is in the form of $\Gamma^a \vdash_a t : T^a\ a$, where a is a binary ability annotation.

Ability Qualifier The *ability qualifier* a in a type judgment $\Gamma^a \vdash_a t : T^a\ a$ describes if the result of evaluating term t is a *resource* – that is, either a reference or a value that may transitively reach other resources. The typing rules ensure that resources are typed with ability qualifier $\top$ wherever they flow.

Ambient Ability The *ambient ability* a of a term t is based on the set of resources from the environment needed to evaluate t. Specifically, it is an upper bound on the ability qualifiers of t's free variables:

$$\Gamma^a(\text{fv}(t)) <: a$$

The definition of computing free variables from a given term is standard, and can be found in Fig. 5. In the typing rules, the key to tie these two notions together is in rule (T-Abs-A): the hypothesis $\Gamma^a(\mathrm{fv}(t) \setminus x) <: a_f$ computes the ability qualifer a_f assigned to the function from the ambient ability of the body expression, *i.e.*, the function becomes itself a resource if it captures any resources (ability qualifier $\neq \bot$) from the environment.

For economy of notation, we often use an augmented term typing judgment that includes the ambient ability a' in the form of $\Gamma^a\ a' \vdash_a t : T^a\ a$.

Syntactic Purity Which expressions are syntactically pure? Returning a resource is not per se impure, but returning a freshly allocated one is (because returning any freshly allocated value is impure), and since there is no distinction in the types, we have to treat any expression with type $T^a\ \top$ as impure.

To determine potential uses of resources as part of evaluation, we have to consider the ambient ability through the free variables of an expression. An expression t with ambient ability $\top$, *i.e.*, $\Gamma^a(\mathrm{fv}(t)) \neq \bot$, is also impure.

Thus, pure expressions are characterized by the augmented typing $\Gamma^a\ \bot \vdash_a t : T^a\ \bot$.

5.2 Metatheory

Similar to λ_e, the core metatheory for λ_a is also standard. We define program equivalence in terms of contextual equivalence and establish it via the binary logical relations for λ_a (details in Section 7).

In particular, we say terms t_1 and t_2 are contextually equivalent, written as $\Gamma^a \models_a t_1 \cong_a t_2 : T^a\ a$, if they have the same observable behavior when placed in any program context C. To achieve that, we use the semantic typing judgment, which is defined as $\Gamma^a \models_a t_1 \approx_{\log_a} t_2 : T^a\ a \stackrel{\text{def}}{=} \forall(\hat{H}, \mathtt{W}) \in G[\![\Gamma^a\ \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)]\!], (\hat{H}, \mathtt{W}, t_1, t_2) \in \mathcal{E}[\![T^a\ a]\!]$, where $\hat{H}$ is a pair of related value environments, $\mathtt{W}$ is a world, $G[\![\Gamma^a\ \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)]\!]$ is the logical interpretation of typing environment, and $\mathcal{E}[\![T^a\ a]\!]$ is the value interpretation of terms, which entails semantic soundness and termination. The detailed definitions and results will be presented in Section 7. The main results are summarized below.

Theorem 5.1 (Fundamental Property). *If* $\Gamma^a \vdash_a t : T^a\ a$, *then* $\Gamma^a \models_a t \approx_{log_a} t : T^a\ a$.

The following theorem shows the soundness of our binary logical relations with respect to contextual equivalence.

Theorem 5.2 (Contextual Equivalence). *if* $\Gamma^a \vdash_a t_1 : T^a\ a$ *and* $\Gamma^a \vdash_a t_2 : T^a\ a$, *then* $\Gamma^a \models_a t_1 \approx_{log_a} t_2 : T^a\ a$ *implies* $\Gamma^a \models_a t_1 \cong_a t_2 : T^a\ a$.

In the following theorem, we write $C : (\Gamma^a; T^a\ a) \Rrightarrow (\Gamma^{a'}; T^{a'}\ a')$ to mean that the context C is a program of type $T^{a'}\ a'$ (closed under $\Gamma^{a'}$) with a hole that can be filled with any program of type $T^{a'}\ a$ (closed under $\Gamma^{a'}$). The typing rules for well-typed contexts can be found in Section 7.

Theorem 5.3 (Effect Safety of Ability Type System). *A syntactically pure expression is semantically (observationally) pure:*

$$\Gamma^a\ \bot \vdash_a t : T^a\ \bot \quad \Rightarrow \quad \forall C : (\Gamma^a; T^a\ \bot) \Rrightarrow (\Gamma^{a'}; T^{a'}\ a).\ \Gamma^{a'} \models_a (\lambda x.\ C[x])\ t \cong_a C[t] : T^{a'}\ a$$

We demonstrate that pure terms have expected algebraic properties in mathematical formulas, *e.g.*, they allow reordering of operand expressions.

Theorem 5.4 (Reordering). *Pure terms can be evaluated in any order,* i.e., *swapping arguments of commutative operations is legal:*

$$\Gamma^a\ \bot \vdash_a t_1 \otimes t_2 : \mathtt{Bool}\ \bot \quad \Rightarrow \quad \Gamma^a \models_a t_1 \otimes t_2 \approx_{log_a} t_2 \otimes t_1 : \mathtt{Bool}\ \bot$$

5.3 Discussion

No Use/Mention Distinction (Negative) We track the flow of resources, but not precisely when they are used. So typing-wise (in a suitable extension with recursive functions), the terms ω and $\lambda x.\ \omega$ are both resources and hence syntactically impure. This may seem like a weakness compared to effect systems, but it is also what enables seamless effect polymorphism.

Effect Polymorphism I (Positive) A notable advantage of capability-based systems is that they provide effect polymorphism without requiring explicit effect quantification. Consider the function `map(f)` again. In our ability type system, the function argument `f` implicitly allows only the effects permitted by the capabilities available to that argument at the call site. As a result, capability systems avoid the annotation burden and complexity typically associated with polymorphic effect systems.

Effect Polymorphism II (Neutral) The preceding discussion concerns passing resources *down* the call chain. In contrast, passing resources *up* the call chain presents different challenges. As mentioned above, we cannot treat a function that may return a resource as pure (because it might return a fresh resource). If we want to return any capability up the chain (even pre-existing), the expression must be treated as impure.

Effect Masking (Positive) The ability system λ_a masks effects that cannot be observed outside of a given expression. For example, the expression $\texttt{let}\ x = \texttt{ref}\ t\ \texttt{in}\ !x$ is syntactically pure in λ_a, as it does not return a resource, and its ambient ability is $\bot$.

Beyond Boolean Refs (Positive) We can generalize mutable references from `Bool` to other non-resource types (*e.g.*, values of closed λ-terms), and still maintain termination and other properties of the system. This would be harder to achieve in an effect system.

6 Type, Ability, and Effect Systems (λ_{ae})

We present λ_{ae}, which integrates ability and effects in a unified system. Then we show that λ_{ae} can encode the effect system λ_e and the ability system λ_a. The meta-theory of λ_{ae} will be presented in Section 7 in the form of binary logical relations.

6.1 Formalization

The typing rules are shown in Fig. 7. The typing judgment is in the form of $\Gamma \vdash t : T \;\; a\ e$, where a is an ability and e is an effect.

Contrary to λ_a presented in Section 5, ability qualifers a now are in the form of $\langle f, s \rangle$, denoting whether the underlying characterized locations may be *fresh* or *stored*. These characterizations have the following meaning:

- *Fresh* means whether an evaluation result may reach new locations allocated during execution. If the value may reach those locations, they must be tracked as they may be accessed by the evaluation context; in this case, the *fresh* component is assigned $\top$. If no allocation occurs during evaluation, or any freshly allocated locations are not reachable from the result (and thus unobservable), the component is assigned $\bot$, and these locations can be safely ignored by the reasoning. See formulas F1 and F2 in the definition of binary logical relations in Section 7.3.
- *Stored* means whether an evaluation result may reach some pre-existing locations. In this case, effects on those locations are externally observable, and thus must be tracked as effect qualifier.

Let $a = \langle\ f,\ s\ \rangle$ be an ability qualifier. From now on, when the context is clear, we write $a.f$ and $a.s$ to refer to its fresh and components, respectively.

This mechanism is exemplified in rules (T-Get-AE) and (T-Put-AE), where effects on fresh cells are ignored, while effects on stored cells are tracked.

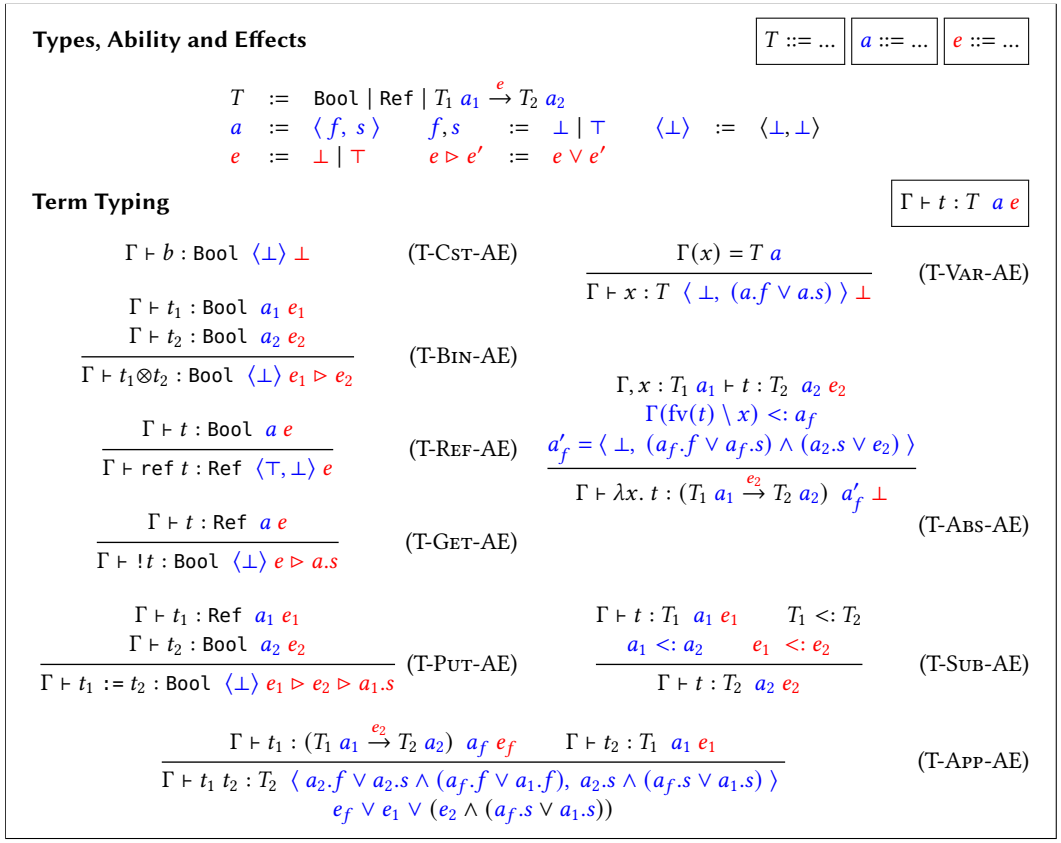

Fig. 7. Ability and Effect Type System λ_{ae}. We write $a.f$, and $a.s$ to refer to the fresh and stored components in ability qualifier a, respectively.

The rule (T-Var-AE) assigns ability qualifier $\langle \bot, (f \vee s) \rangle$, since variables references are by definition to a pre-existing value, thus they cannot be fresh. Freshness is reserved for allocations during evaluation of a term, but during evaluation of a variable reference no allocation happens.

The function application rule (T-App-AE) ensures that if a function has a latent effect e_2, then either the function itself or the argument carries capabilities. These are the only values the function has access to, thus any locations it may touch as part of its effect must be reachable through those values.

The function abstraction rule (T-TBS-AE) specifies that a function is treated as a resource only if its value carries either existing or fresh capabilities, and either has latent effect or returns a value that carries capabilities from the environment.

Fig. 8 shows the subtyping rules for λ_{ae}. The rule S-Sub-AE defines the general sub-typing relation for ability qualifiers, where subtyping is applied componentwise to each of the qualifier's components. For each component, the subtyping relations are defined by the rules S-A-AE, S-A⊥-AE and S-A⊤-AE. The rule for function types (S-Abs-AE) demand contravariance in the argument type and the effects, and covariance in the result type.

Syntactic Purity Which expressions are syntactically pure? In our λ_{ae} system, a term assigned ability and effect qualifier $\langle f, s \rangle$ $\top$ is impure, as the effect qualifier $\top$ means evaluating the term may induce observable effects. In the case of a function application t_1 t_2, if the reduction of t_1 yields a function value whose ambient ability is $\langle \bot \rangle$, then the effect of executing the function body may

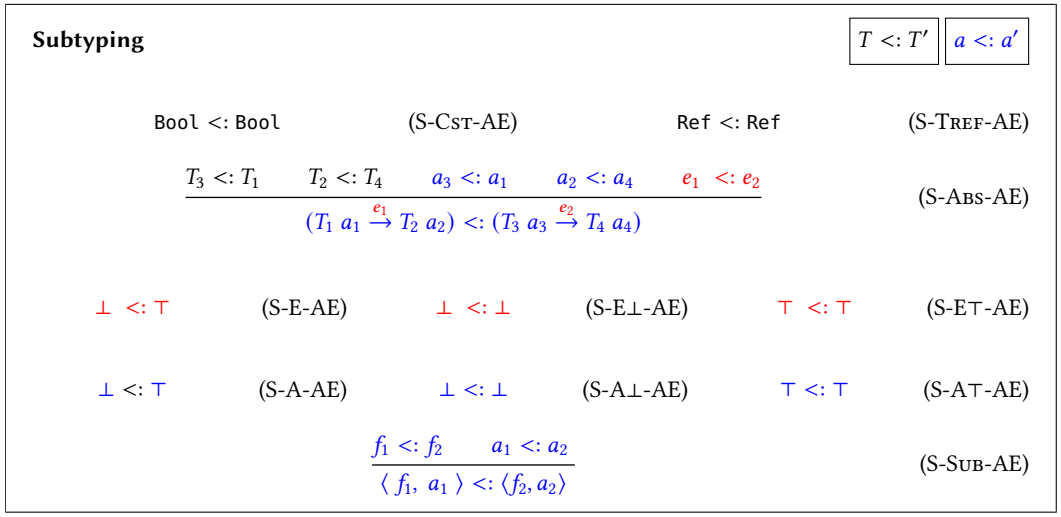

Fig. 8. Ability and Effect Subtyping System λ_{ae}.

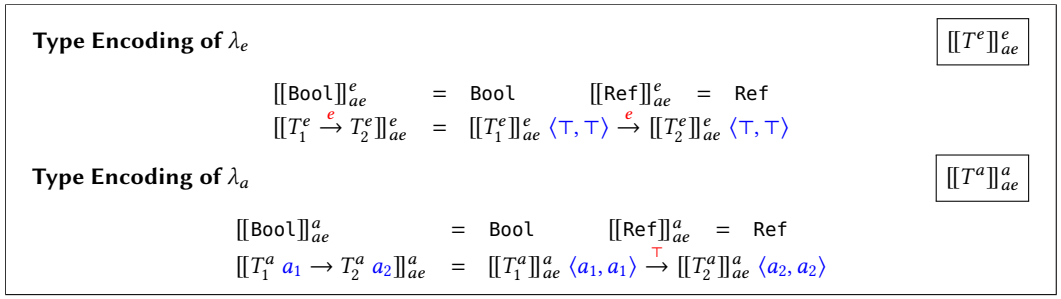

Fig. 9. Type translation from λ_e and λ_a to λ_{ae} respectively.

be refined to $\bot$, despite the body carrying a latent effect $\top$, as in rule T-APP-AE of Fig. 7 specifies. A term assigned qualifier $\langle \top, s \rangle\ \bot$ is also impure. The reason is that while evaluating the term does not induce observable effects on pre-existing store locations (*i.e.*, $\bot$), it yields a freshly allocated location, which may be observed by the execution context (*i.e.*, $\top$). A term assigned qualifier $\langle \bot, \top \rangle\ \bot$ means that its evaluating result carries abilities, as the qualifier means that evaluating the term does not yield freshly allocation locations (*i.e.*, $\bot$), or induces observable effects ($\bot$), but it is granted the ability to access pre-existing locations (*i.e.*, $\top$). The term is pure if its ambient ability is $\langle \bot \rangle$.

6.2 Encoding of Pure Effect and Pure Ability System

Now we show that λ_{ae} can encode the effect system λ_e and the ability system λ_a.

Fig. 9 (top) shows the type encoding of λ_e. The base types are encoded directly. For function types, as the argument and return types in λ_e do not have any information about ability, the encoding conservatively assumes the $\top$ for the fresh and stored components in their translated types.

Fig. 9 (bottom) shows the type encoding of λ_a. The base types are encoded directly. For function types, we directly translate the ability qualifier for arguments and return values in λ_a into the target type. As the capability system does not have information on effects, the encoding conservatively assumes the $\top$.

We define $[\![\Gamma^e]\!]^e_{ae}$ and $[\![\Gamma^a]\!]^a_{ae}$ that translate the source typing environments, Γ^e and Γ^a, to the target ones by mapping each type according to the transition rules defined in Fig. 9.

```
let y = true in                  // : Bool ⟨⊥⟩   in context [ y: Bool ⟨⊥⟩ ]
let a = new Ref(false) in        // : Ref ⟨⊥,⊤⟩ in context [ a: Ref ⟨⊤,⊥⟩, x: Bool ⟨⊥⟩ ]
def id (x: Bool) = { x }         // : (Bool ⟨⊥⟩   =>⊥ Bool ⟨⊥⟩) ⟨⊥⟩
def idr (x: Ref) = { x }         // : (Ref ⟨⊤,⊥⟩ =>⊥ Ref ⟨⊥,⊤⟩) ⟨⊥⟩
def alloc (x: Bool) = { ref x }  // : (Bool ⟨⊥⟩   =>⊥ Ref ⟨⊤,⊥⟩) ⟨⊥⟩
def leak: (x: Bool) = { a }      // : (Bool ⟨⊥⟩   =>⊥ Ref ⟨⊥,⊤⟩) ⟨⊥,⊤⟩
def use (x: Ref) = { !a }        // : (Ref ⟨⊤,⊥⟩ =>⊤ Bool ⟨⊥⟩) ⟨⊥,⊤⟩
def usearg (x: Ref) = { !x }     // : (Ref ⟨⊥,⊤⟩ =>⊤ Bool ⟨⊥⟩) ⟨⊥⟩
def mention (x: Ref) = { true }  // : (Ref ⟨⊥,⊤⟩ =>⊥ Bool ⟨⊥⟩) ⟨⊥⟩
def capture (x: Ref) = { λ y:Ref. !a }
                                 // : (Ref ⟨⊤,⊥⟩ =>⊥ ((Ref ⟨⊤,⊥⟩ =>⊤ Bool ⟨⊥⟩) ⟨⊥,⊤⟩)) ⟨⊥,⊤⟩
id(y)                            // : Bool ⟨⊥⟩ ⊥
idr(a)                           // : Ref ⟨⊥,⊤⟩ ⊥
use(y)                           // : Bool ⟨⊥⟩ ⊤
usearg(a)                        // : Bool ⟨⊥⟩ ⊤
mention(a)                       // : Ref ⟨⊥,⊤⟩ ⊥
```

Fig. 10. Selected examples in Fig. 2 that are annotated with abilities and effects.

The following two lemmas show the encodings are sound, *i.e.*, if a term is well-typed in the source system, then its translation is well-typed in the target system under the translated environments.

Theorem 6.1. *If* $\Gamma^e \vdash_e t : T^e\ e$ *then* $[[\Gamma^e]]^e_{ae} \vdash t : [[T]]^e_{ae}\ \langle \top, \top \rangle\ e$.

Theorem 6.2. *If* $\Gamma^a \vdash_a t : T^a\ a$ *then* $[[\Gamma^a]]^a_{ae} \vdash t : [[T]]^a_{ae}\ \langle a, a \rangle\ \top$.

6.3 Discussion

Fig. 10 shows selected examples in Fig. 2 that are annotated with abilities and effects in λ_{ae}. In the absence of type abstraction, we introduce two separate functions for the identity function: `id` for Boolean arguments and `idr` for Boolean reference arguments. Invoking them, *i.e.*, `id(y)` and `idr(a)` yields proper abilities and effects. The function `alloc` creates a new reference initialized with the given Boolean argument. The allocation does not induce observable effects, but the resulting reference carries an ability $\langle \bot, \top \rangle$, meaning that the return value is fresh. In contrast, the function `leak` returns the reference captured from the environment, which is not fresh. Thus, its ability $\langle \top, \bot \rangle$. The functions `use` and `usearg` induce effects on the captured reference from the environment and its argument respectively. Thus, their effects are $\top$. Note that the ability of `use`'s argument can be $\langle \bot \rangle$, as the function itself does not use its argument. However, `use(a)` becomes untypable. In contrast, the function `mention` does not use its argument's ability or capture anything from the environment; hence, its effect is $\bot$. In addition, the functions `leak` and `use` capture the ability carried by the reference `a` from the environment, thus their abilities are $\langle \bot, \top \rangle$; others are just $\langle \bot \rangle$.

7 Contextual Equivalence and Purity via Binary Logical Relations for λ_{ae}

In this section, we present the details of the earlier theorems, which are proved using binary logical relations that show semantically equivalent values and terms are logically related.

7.1 Contextual Equivalence

We write $\Gamma \models t_1 \cong t_2 : T\ a\ e$ to mean that programs t_1 and t_2 are *contextually equivalent* at type $T\ a\ e$, meaning that they exhibit indistinguishable behavior in all program contexts C with a hole of type $T\ a\ e$. That is to say if $C[t_1]$ exhibits some observable behavior, then so does $C[t_2]$.

Following the approach of Timany et al. [2024] and related prior work [Ahmed et al. 2009; Bao et al. 2025], we define a judgement for logical equivalence using binary logical relations, written as $\Gamma \models t_1 \approx_{\log} t_2 : T\ a\ e$.

Context for Contextual Equivalence

$$C \;::=\; \square \mid C\ t \mid t\ C \mid \lambda x.\ C \mid \texttt{ref}\ C \mid\ !C \mid C := t \mid t := C \mid t_1 \otimes C \mid C \otimes t_2$$

Context Typing Rules $\boxed{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T'\ a'\ e'\ \varphi')}$

$$\square : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma; T\ a\ e\ \varphi) \tag{C-$\square$}$$

$$\frac{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow ((\Gamma', x : T_1\ a_1); T_2\ a_2\ e_2\ \varphi') \qquad \Gamma'(\varphi' \setminus x) <: a_f \qquad a'_f = \langle\ \bot,\ (a_f.s \vee a_f.f) \wedge (a_2.s \vee e_2)\ \rangle}{\lambda x.\ C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow \Gamma'; (T_1\ a_1 \xrightarrow{e_2} T_2\ a_2)\ a'_f\ \bot\ (\varphi' \setminus x)} \tag{C-λ}$$

$$\frac{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; (T_1\ a_1 \xrightarrow{e} T_2\ a_2)\ a_f\ e_f\ \varphi') \qquad \Gamma' \vdash t : T_1\ \ a_1\ e_1}{C\ t : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T_2\ \langle\ a_2.f \vee a_2.s \wedge (a_f.f \vee a_1.f),\ a_2.s \wedge (a_f.s \vee a_1.s)\ \rangle\ e_f \rhd e_1 \rhd (e \wedge (a_f.s \vee a_1.s))\ \varphi')} \tag{C-APP-1}$$

$$\frac{\Gamma' \vdash t : T_1\ a_1 \xrightarrow{e} T_2\ a_2\ \ a_f\ e_f \qquad C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T_1\ a_1\ e_1\ \varphi_1)}{t\ C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T_2\ \langle\ a_2.f \vee a_2.s \wedge (a_f.f \vee a_1.f),\ a_2.s \wedge (a_f.s \vee a_1.s)\ \rangle\ e_f \rhd e_1 \rhd (e \wedge (a_f.s \vee a_1.s))\ \mathrm{fv}(t))} \tag{C-APP-2}$$

$$\frac{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ a'\ e'\ \varphi')}{\texttt{ref}\ C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Ref}\ \langle\ \top,\ \bot\ \rangle\ e'\ \varphi')} \tag{C-REF}$$

$$\frac{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Ref}\ a'\ e'\ \varphi')}{!C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}; \langle\bot\rangle\ e' \rhd a'.s\ \varphi')} \tag{C-GET}$$

$$\frac{C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Ref}\ a_1\ e_1\ \varphi_1) \qquad \Gamma \vdash t : \texttt{Bool}\ \ a_2\ e_2}{C := t : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ \langle\bot\rangle\ e_1 \rhd e_2 \rhd a_1.s\ \varphi_1)} \tag{C-PUT-1}$$

$$\frac{\Gamma \vdash t : \texttt{Ref}\ \ a_1\ e_1 \qquad C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ a_2\ e_2\ \varphi_2)}{t := C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ \langle\bot\rangle\ e_1 \rhd e_2 \rhd a_1.s\ \varphi_2)} \tag{C-PUT-2}$$

$$\frac{\Gamma \vdash t : \texttt{Bool}\ \ a_1\ e_1 \qquad C; (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ a_2\ e_2\ \varphi_2)}{t \otimes C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ \langle\bot\rangle\ e_1 \rhd e_2\ \varphi_2)} \tag{C-BIN-1}$$

$$\frac{C; (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ a_1\ e_1\ \varphi_1) \qquad \Gamma \vdash t : \texttt{Bool}\ \ a_2\ e_2}{C \otimes t : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; \texttt{Bool}\ \langle\bot\rangle\ e_1 \rhd e_2\ \varphi_1)} \tag{C-BIN-2}$$

Fig. 11. Selected Context Typing Rules.

Differing from reduction contexts, contexts C for reasoning about equivalence allow a "hole" to appear in any place. We write $C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T'\ a' e'\ \varphi')$ to mean that the context C is a program of type $T'\ a'\ e'\ \varphi'$ (closed under Γ) with a hole that can be filled with any program of type $T\ a\ e\ \varphi$ (closed under Γ). The type rules for well-typed contexts imply that if $\Gamma \vdash t : T\ \ a\ e$, $\varphi = \mathrm{fv}(t)$, $\varphi' = \mathrm{fv}(C[t])$ and $C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\Gamma'; T'\ a'\ e'\ \varphi')$ hold, then $\Gamma' \vdash C[t] : T'\ \ a'\ e'$. Selected typing rules for well-typed contexts can be found in Fig. 11.

We adopt the standard definition of contextual equivalence [Ahmed et al. 2009], as follows:

Definition 7.1 (Contextual Equivalence). Term t_1 is *contextually equivalent* to t_2, written $\Gamma \models t_1 \cong t_2 : T\ a\ e$, if $\Gamma \vdash t_1 : T\ \ a\ e$, and $\Gamma \vdash t_2 : T\ \ a\ e$, and $\varphi = \mathrm{fv}(t)$, and $\forall C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\emptyset; \texttt{Bool}\ \langle\bot\rangle\ \bot\ \emptyset)$. $C[t_1]\downarrow \iff C[t_2]\downarrow$.

Store Typing, Relational Worlds, Well-Defined Stores λ_{ae}

$$\Sigma := \varnothing \mid \Sigma, \ell : \{\mathtt{true}, \mathtt{false}\}$$

$$\mathtt{W} \equiv_{(\mathsf{L}_1\mathsf{L}_2)} \mathtt{W}' = (\forall\, \ell_1 \in \mathsf{L}_1, \ell_2 \in \mathsf{L}_2.\, \mathtt{W}_f(\ell_1, \ell_2) \Rightarrow \mathtt{W}'_f(\ell_1, \ell_2))$$

$$\boxed{(\sigma_1, \sigma_2) : \mathtt{W}}^{\mathsf{L}_1}_{\mathsf{L}_2} \stackrel{\text{def}}{=} \sigma_1 : \mathtt{W}_1 \wedge \sigma_2 : \mathtt{W}_2 \wedge (\forall (\ell_1, \ell_2) \in \mathtt{W}.\, \ell_1 \in \mathsf{L}_1 \wedge \ell_2 \in \mathsf{L}_2 \Rightarrow (\exists\, b.\, \sigma_1(\ell_1) = \sigma_2(\ell_2) = b))$$

Fig. 12. Definitions of store typing, relational words and well-defined stores with respect to a world.

Reachable Locations λ_{ae}

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

Fig. 13. Computing reachable locations from a given value.

We write $t \downarrow$ to mean term t terminates, if $\emptyset,\ \emptyset,\ t \Downarrow \sigma,\ v$, for some value v and final store σ. This standard definition characterizes partial program equivalence, where program termination serves as the observable behavior. However, since we focus on a total fragment of the systems here, termination alone is insufficient as an observer for program equivalence. We thus rely on a refined definition of contextual equivalence using Boolean contexts.

$$\forall C : (\Gamma; T\ a\ e\ \varphi) \Rrightarrow (\emptyset; \mathtt{Bool}\ \langle\bot\rangle\ \bot\ \emptyset).\, \exists\, \sigma, \sigma', v.\, \emptyset,\ \emptyset,\ C[t_1] \Downarrow v,\ \sigma \wedge \emptyset,\ \emptyset,\ C[t_2] \Downarrow v,\ \sigma'.$$

That is to say, we consider two terms contextually equivalent if they yield the same answer value in all Boolean contexts.

7.2 The World Model

Definition 7.2 (World). A world $\mathtt{W}$ is a triple of (Σ_1, Σ_2, f), where

- Σ_1 and Σ_2 are store typings defined in Fig. 12,
- $f \subseteq (\mathrm{dom}(\Sigma_1) \times \mathrm{dom}(\Sigma_2))$ is a partial bijection.

A world defines relational stores. The partial bijection captures the fact that a relation holds under permutation of store locations.

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

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

We write $\mathrm{dom}_1(\mathtt{W})$ and $\mathrm{dom}_2(\mathtt{W})$ to refer the first and second components in the world $\mathtt{W}$, *i.e.*, $\mathrm{dom}_1(\mathtt{W}) = \mathrm{dom}(\Sigma_1)$ and $\mathrm{dom}_2(\mathtt{W}) = \mathrm{dom}(\Sigma_2)$. Let $\mathtt{W}$ and $\mathtt{W}$ be two worlds. We write $\mathrm{dom}(\mathtt{W}) \subseteq \mathrm{dom}(\mathtt{W}')$ to mean $\mathrm{dom}_1(\mathtt{W}) \subseteq \mathrm{dom}_1(\mathtt{W}') \wedge \mathrm{dom}_2(\mathtt{W}) \subseteq \mathrm{dom}_2(\mathtt{W}')$. We write $\mathtt{W}_f$ to mean the third component in the world $\mathtt{W}$.

If $\mathtt{W}$ and $\mathtt{W}'$ are worlds, such that $\mathrm{dom}_1(\mathtt{W}) \cap \mathrm{dom}_1(\mathtt{W}') = \mathrm{dom}_2(\mathtt{W}) \cap \mathrm{dom}_2(\mathtt{W}') = \emptyset$, then $\mathtt{W}$ and $\mathtt{W}'$ are called disjoint, and we write $\mathtt{W};\mathtt{W}'$ to mean extending $\mathtt{W}$ with a disjoint world $\mathtt{W}'$. Let σ_1 and σ_2 be two stores. We write $\boxed{(\sigma_1, \sigma_2) : \mathtt{W}}^{L_1}_{L_2}$ to mean the stores are well-defined with respect to the world $\mathtt{W}$ at a pair of locations (L_1, L_2), which is formally defined in Fig. 12.

Reachable Locations Following Bao et al. [2025]'s work, the definition of logical relations needs to know the reachable locations from a given value v, which is computed by the function $L(v)$, as shown in Fig. 13. Boolean values, b, reach the empty set of locations. A location ℓ can only reach itself. Thus, its reachable set is the singleton set $\{\ell\}$. The set of locations that are reachable from a function value $\lambda x.\ t$ are the set of the locations reachable from the values retrieved from its free variables in the value environment H, *i.e.*, $L_H(\mathrm{fv}(\lambda x.\ t))$.

Relational Worlds Instead of using the standard definition of world extension, we adapt the relational worlds from Bao et al. [2025]'s work, which leverages reachable locations to achieve a

degree of localization. In Fig. 12, relational worlds, written as $\mathtt{W} \equiv_{(\mathtt{L}_1\mathtt{L}_2)} \mathtt{W}'$, parameterize the definition of world extension with related reachable locations $\mathtt{L}_1$ and $\mathtt{L}_2$. The definition is symmetric, allowing $\mathtt{W}'$ to be defined from $\mathtt{W}$, such that they agree on $\mathtt{L}_1$ and $\mathtt{L}_2$, leaving the remainder unspecified. The standard definition of store typing extension $\mathtt{W} \sqsubseteq \mathtt{W}'$ can be defined as $\mathtt{W} \equiv_{(\mathrm{dom}_1(\mathtt{W})\mathrm{dom}_2(\mathtt{W}))} \mathtt{W}'$. The relational worlds are used in reasoning about functions. See detailed explanation in Section 7.4.

7.3 Use vs Mention Aware Binary Logical Relations

We first introduce the interpretation of typing context, which establishes the assumption of the reasoning.

Definition 7.3 (Semantic Typing Context). The definition of typing context interpretation is defined as follows:

$$
\begin{aligned}
G[\![\Gamma\ \varphi\ u]\!] \ = \ \{(\mathtt{W}, \hat{H}, \hat{R}) \mid\ & \mathrm{dom}(\hat{H}_i) = \mathrm{dom}(\hat{R}_i) = \mathrm{dom}(\Gamma) \wedge \\
& (\forall x, T, a.\, \Gamma(x) = T\ a \Rightarrow\ \exists\, ux, L_1, L_2.\, ux = (\neg a \vee u) \wedge \\
& \qquad (x \in \varphi \Rightarrow (\mathtt{W}, \hat{H}_1(x), \hat{H}_2(x), ux, L_1, L_2) \in \mathcal{V}[\![T]\!]) \wedge \\
& \qquad ((\neg a \vee u) \Rightarrow \hat{R}_1(x) = L_1 \wedge \hat{R}_2(x) = L_2) \wedge \\
& \qquad ((\neg a \vee \neg u) \Rightarrow (L_1 = \emptyset\ \wedge\ L_2 = \emptyset)))\}.
\end{aligned}
$$

Semantic typing context is defined as a set of tuples of the form $(\mathtt{W}, \hat{H}, \hat{R})$, where $\mathtt{W}$ is a world defined in Def. 7.2. The component $\hat{H}$ 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 $\mathrm{dom}(\hat{H}_1)$ and $\mathrm{dom}(\hat{H}_2)$ to mean the domain of the first and second value environment respectively. The component $\hat{R}$ ranges over a pair of relational reachability environments that are finite maps from variables x to pairs of locations (L_1, L_2), which are reachable from their respectively referred values, *i.e.*, $\hat{H}_1(x)$ and $\hat{H}_2(x)$. The domains of $\hat{R}_1$ and $\hat{R}_2$ are defined similarly to those of $\hat{H}$.

The reachability information defined in $\hat{R}$ are refined with respect to a reasoning mode, which allows reasoning to distinguish whether external resource is actually used or only mentioned in two pairs of terms, with respect to the effect qualifier. The reasoning mode is determined by the binary semantic typing judgement.

Definition 7.4 (Semantic Typing Judgment). The semantic typing judgement is defined as:

$$
\begin{aligned}
\Gamma \models t_1 \approx_{\log} t_2 : T\ a\ e \stackrel{\text{def}}{=}\ & \forall u.(e \Rightarrow u) \Rightarrow\ (\forall \mathtt{W}, \hat{H}, \hat{R}. \\
& (\mathtt{W}, \hat{H}, \hat{R}) \in G[\![\Gamma\ (\mathrm{fv}(t_1) \cup \mathrm{fv}(t_2))\ u]\!].\ (\mathtt{W}, \hat{H}, t_1, t_2, u) \in \mathcal{E}[\![T\ a\ e]\!]).
\end{aligned}
$$

In the above, the boolean parameter u means that the current reasoning is in a use mode if it is `true`; otherwise it is in a mention mode, meaning that the current reasoning does not require the knowledge about any external resource. Here, we write $e \Rightarrow u$ to mean $e = \top \Rightarrow u = \mathtt{true}$. The condition dictates that the reasoning must be in a use mode (*i.e.*, $u = \mathtt{true}$) whenever the computation induces observable effects (*i.e.*, $e = \top$).

Example Fig. 10 shows the examples annotated with abilities and effects. The programs `id(y)` and `mention(a)` can be reasoned in either mode, (*i.e.*, $u = \mathtt{true}$ or $u = \mathtt{false}$), as they don't have observable effects (*i.e.*, $e = \bot$). In contrast, the programs `use(y)` and `usearg(a)` can only be reasoned in the use mode (*i.e.*, $u = \mathtt{true}$), as they induce observable effect (*i.e.*, $e = \top$), and require access to the reference `y`. However, the function `usearg` can be reasoned in the mention mode by using Lemma 7.5 (shown below).

Properties The following lemma allows reasoning about a sub-term in the mention mode, if the overall reasoning is in a use mode.

Binary Value Interpretation of Types and Terms λ_{ae}

$$\sigma \rightarrow_L \sigma' = \forall \ell \in \mathrm{dom}(\sigma).\ \ell \notin L \Rightarrow \sigma(\ell) = \sigma'(\ell)$$
$$\mathtt{fr}(\sigma', \sigma) = \mathrm{dom}(\sigma') - \mathrm{dom}(\sigma)$$

$$\mathcal{V}[\![\mathtt{Bool}]\!] = \{(\mathtt{W}, b, b, u, \mathsf{L}_1, \mathsf{L}_2)\}$$
$$\mathcal{V}[\![\mathtt{Ref}]\!] = \{(\mathtt{W}, \ell_1, \ell_2, u, \mathsf{L}_1, \mathsf{L}_2) \mid u \Rightarrow (\mathtt{W}(\ell_1, \ell_2) \wedge \ell_1 \in \mathsf{L}_1 \wedge \ell_2 \in \mathsf{L}_2)\}$$

$$\begin{aligned}\mathcal{V}[\![T_1\ a_1 \xrightarrow{e} T_2\ a_2]\!] = \{&(\mathtt{W}, \langle H_1,\ \lambda x.\ t_1\rangle, \langle H_2,\ \lambda x.\ t_2\rangle, u, \mathsf{L}_{\lambda_1}, \mathsf{L}_{\lambda_2}) \mid \forall \sigma_1', \sigma_2', \mathtt{W}', \mathsf{L}_1, \mathsf{L}_2, v_1, v_2, \mathsf{L}_{v_1}, \mathsf{L}_{v_2}.\\
&\forall ux, ubdy, ur.\ \boxed{\mathrm{L1}}\ ((e \vee ur \wedge a_2.s) \Rightarrow \mathtt{W} \equiv_{(\mathsf{L}_{\lambda_1}\mathsf{L}_{\lambda_2})} \mathtt{W}') \wedge\\
&((e \vee ur \wedge a_2.s) \Rightarrow (ux \wedge u)) \wedge (e \Rightarrow ubdy) \wedge (ur = \neg a_1.s \vee ubdy) \wedge\\
&(\neg a_1 \Rightarrow ux) \wedge \mathrm{dom}(\mathtt{W}) \subseteq \mathrm{dom}(\mathtt{W}') \wedge \boxed{\mathrm{S1}}\ \boxed{(\sigma_1', \sigma_2') : \mathtt{W}'}^{\mathsf{L}_1}_{\mathsf{L}_2} \wedge\\
&\boxed{\mathrm{R1}}\ (e \Longrightarrow \mathsf{L}_{v_i} \cup \mathsf{L}_{\lambda_i} \subseteq \mathsf{L}_i) \wedge \boxed{\mathrm{R2}}\ ((\neg a_1 \vee \neg ux) \Rightarrow (\mathsf{L}_{v_i} = \emptyset)) \wedge\\
&(\mathtt{W}', v_1, v_2, ux, \mathsf{L}_{v_1}, \mathsf{L}_{v_2}) \in \mathcal{V}[\![T_1]\!] \Rightarrow\\
&(\exists \sigma_1'', \sigma_2'', \mathtt{W}'', r_1, r_2, \mathsf{L}_{r_1}, \mathsf{L}_{r_2}.\ \mathtt{W}' \sqsubseteq \mathtt{W}'' \wedge H_1; (x, v_1),\ \sigma_1',\ t_1 \Downarrow \sigma_1'',\ r_1 \wedge\\
&\quad H_2; (x, v_2),\ \sigma_2',\ t_2 \Downarrow \sigma_2'',\ r_2 \wedge \mathrm{dom}(\sigma_i') \subseteq \mathrm{dom}(\sigma_i'') \wedge\\
&\quad \boxed{\mathrm{S2}}\ \boxed{(\sigma_1'', \sigma_2'') : \mathtt{W}''}^{\mathsf{L}_1 \cup \mathtt{fr}(\sigma_1'', \sigma_1')}_{\mathsf{L}_2 \cup \mathtt{fr}(\sigma_2'', \sigma_2')} \wedge (\neg ur \Rightarrow L_{r_i} = \emptyset) \wedge\\
&\quad (\mathtt{W}'', r_1, r_2, u_r, \mathsf{L}_{r_1}, \mathsf{L}_{r_2}) \in \mathcal{V}[\![T_2]\!] \wedge\\
&\quad \boxed{\mathrm{R3}}\ \mathsf{L}_{r_i} \subseteq \emptyset \cup_{a_2.s} (\mathsf{L}_{v_i} \cup \mathsf{L}_{\lambda_i}) \cup_{a_2.f} \mathtt{fr}(\sigma_i'', \sigma_i') \wedge\\
&\quad \boxed{\mathrm{E1}}\ \sigma_i' \rightarrow_{\emptyset \cup_e (\mathsf{L}_{\lambda_i} \cup \mathsf{L}_{v_i})} \sigma_i'' \wedge \boxed{\mathrm{F1}}\ (\neg a_2.f \Rightarrow \mathtt{W}_f'' = \mathtt{W}_f')))\}\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![T\ a\ e]\!] = \{&(\mathtt{W}, \hat{H}, t_1, t_2, u) \| \forall \sigma_1, \sigma_2, \mathsf{L}_1, \mathsf{L}_2, \boxed{(\sigma_1, \sigma_2) : \mathtt{W}}^{\mathsf{L}_1}_{\mathsf{L}_2} \wedge \emptyset \cup_e L_{\hat{H}_i}(\mathrm{fv}(t_i)) \subseteq \mathsf{L}_i \wedge\\
&\exists \sigma_1', \sigma_2', \mathtt{W}', u', v_1, v_2, \mathsf{L}_{v_1}, \mathsf{L}_{v_2}.\ \hat{H}_1,\ \sigma_1,\ t_1 \Downarrow \sigma_1',\ v_1 \wedge \hat{H}_2,\ \sigma_2,\ t_2 \Downarrow \sigma_2',\ v_2 \wedge\\
&\quad \mathtt{W} \sqsubseteq \mathtt{W}' \wedge \mathrm{dom}(\sigma_i') \subseteq \mathrm{dom}(\sigma_i) \wedge \boxed{(\sigma_1', \sigma_2') : \mathtt{W}'}^{\mathsf{L}_1 \cup \mathtt{fr}(\sigma_1', \sigma_1)}_{\mathsf{L}_2 \cup \mathtt{fr}(\sigma_2', \sigma_2)} \wedge\\
&\quad (\mathtt{W}', v_1, v_2, u', \mathsf{L}_1, \mathsf{L}_2) \in \mathcal{V}[\![T]\!] \wedge u' = (\neg a \vee u) \wedge (\neg u \Rightarrow L_{v_i} = \emptyset) \wedge\\
&\quad L(v_i) \subseteq \emptyset \cup_{a.s} L_{\hat{H}_i}(\mathrm{fv}(t_i)) \cup_{a.f} \mathtt{fr}(\sigma_i', \sigma_i) \wedge\\
&\quad \boxed{\mathrm{E2}}\ \sigma_i \rightarrow_{\emptyset \cup_e L_{\hat{H}_i}(\mathrm{fv}(t_i))} \sigma_i' \wedge \boxed{\mathrm{F2}}\ (\neg a.f \Rightarrow \mathtt{W}_f' = \mathtt{W}_f)\}\end{aligned}$$

Fig. 14. The binary interpretation of types and terms, and semantic context interpretation. The notation $\neg a$ is a shorthand for $\neg(a.f \vee a.s)$, meaning that a value does not reach any resource.

Lemma 7.5. *If* $(\mathtt{W}, \hat{H}, \hat{R}) \in G[\![\Gamma\ \varphi\ u]\!]$ *and* u' *implies* u, *then* $(\mathtt{W}, \hat{H}, \hat{R}) \in G[\![\Gamma\ \varphi\ u']\!]$.

The following lemma allows the established equivalent terms in the mention mode to remain equivalent in the use mode, provided they induce no observable effects and carry no abilities.

Lemma 7.6. *If* $(\mathtt{W}, \hat{H}, t_1, t_2, \mathtt{false}) \in \mathcal{E}[\![T\ a\ e]\!]$, *and* $e = \bot$, *and* $a = \langle\bot\rangle \vee u = \mathtt{false}$, *then* $(\mathtt{W}, \hat{H}, t_1, t_2, u) \in \mathcal{E}[\![T\ a\ e]\!]$.

7.4 Value Interpretation of Types

The value interpretation of types are defined in Fig. 14. In the figure, we use the subscript i in formulas to refer to both terms when the context is clear, where $i = 1$ or 2. For example, in the value interpretation of function types, we write $\mathsf{L}_{v_i} = \emptyset$ to mean $\mathsf{L}_{v_1} = \emptyset \wedge \mathsf{L}_{v_2} = \emptyset$.

To simplify notation and improve readability, we conflate the values of $\top/\top$ and $\bot/\bot$ with logical truth and falsehood, respectively. For example, we write $e \Rightarrow P$ (for some effect e and predicate P)

to mean $e = \top \Rightarrow P$. In addition, we use the shorthand $\neg a$ to express $\neg(a.f \vee a.s)$, meaning that a value does not reach any resource.

In Fig. 14, 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.

The value interpretation of types, written as $\mathcal{V}[\![T]\!]$, is a set of tuples of form $(\mathtt{W}, v_1, v_2, u, \mathtt{L}_1, \mathtt{L}_2)$, where v_1 and v_2 are values, and $\mathtt{W}$ is a world, u denotes whether the pair of values can be used, and $\mathtt{L}_1$ and $\mathtt{L}_2$ are some uppper bound of the locations that the reasoning cares.

Boolean Type. A pair of boolean values are related if they are both `true` or `false`. Since operations on boolean values do not need any resources, they can be used if the associated u component allows.

Reference Type. A pair of locations (ℓ_1, ℓ_2) are related with respect to the world $\mathtt{W}$ if they are used by the reasoning (*i.e.*, u holds). In this case, their relation must be defined in the world $\mathtt{W}$, and their reachable locations have to be bounded by $\mathtt{L}_1$ and $\mathtt{L}_2$ respectively. Otherwise (*i.e.*, u does not hold), the reasoning does not have any knowledge about the pair.

Function Types. The set of values of function type $T_1\ a_1 \xrightarrow{e} T_2\ a_2$ with respect to the world $\mathtt{W}$ is defined based on its computational behavior. Note that we inline the term interpretation to characterize the function body's computational behavior.

Safe Execution. As mentioned earlier, we adopt relational worlds from Bao et al. [2025]'s work to characterize localization. For example, if a function either has a latent effect, or the result carries stored ability (*i.e.*, $e \vee ur \wedge a_2.s$), then a safe execution of the function body needs to access locations reachable from the function (formulas L1 and S1). Here, the formula $\mathtt{W} \equiv_{(L_{\lambda_1} L_{\lambda_2})} \mathtt{W}'$ means that types are only preserved with respect to reachable locations from function values.

Thus, the reasoning must need the locations reachable from the function. In another word, if the function has no latent effect, or its return value does not reach any pre-existing locations (*i.e.*, $a_2.s = \bot$), then the function can be safely executed in any future world extending from the current one $\mathtt{W}$ where the argument values are valid.

From Abilities and Effects to Reachable Locations. Formulas R1, R2 and R3 capture properties of reachable locations enabled by the ability and effect qualifiers. Formula R1 states that if the function has a latent effect, then the reachable locations from the function and its argument are bounded by the locations that the reasoning cares (*i.e.*, $\mathtt{L}_i$). Formula R2 states that the argument cannot reach any locations if the fresh and stored component of the argument's ability qualifier are both $\bot$ or the argument is not used. Formula R3 states that if the return value's stored component (in its ability qualifier) is $\top$, then it may reach locations from the function or its argument. In addition, if its fresh component is $\top$, it may reach freshly allocated locations.

From Effects to Value Preservation. Formulas E1 and E2 capture the value preservation over a potentially effectful computation, *i.e.*, values stored in pre-existing locations that are not covered by the effects remain unchanged.

Terms. In addition to the properties about reachable locations, safe execution and value preservation, which are similar to those for function bodies, two terms are related if their reduction results are related at given types.

7.5 The Compatibility Lemmas, Fundamental Theorem and Soundness

Fig. 14 defines the interpretation of typing context. 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 $\mathrm{dom}_1(\hat{H})$ and $\mathrm{dom}_2(\hat{H})$ to mean the domain of the first and second value environment respectively.

The semantic typing rules have the same shape as the corresponding syntactic ones, but turning the symbol $\vdash$ in Fig. 7 into $\models$. Each of the semantic typing rules is a compatibility lemma and has been proved in Coq.

Theorem 7.7 (Fundamental Property). *If* $\Gamma \vdash t : T \;\; a\ e$, *then* $\Gamma \models t \approx_{log} t : T \;\; a\ e$.

Theorem 7.8 (Soundness with respect to Contextual Equivalence). *The binary logical relation is sound with respect to contextual equivalence,* i.e., *if* $\Gamma \vdash t_1 : T \;\; a\ e$, *and* $\Gamma \vdash t_2 : T \;\; a\ e$, *then* $\Gamma \models t_1 \approx_{log} t_2 : T \;\; a\ e$ *implies* $\Gamma \models t_1 \cong t_2 : T \;\; a\ e$.

Theorem 7.9 (Effect Safety of Ability and Effect Type Systems). *A syntactically pure expression is semantically (observationally) pure:*

$$\Gamma \vdash t : T \;\; a\ \bot \Rightarrow \forall C : (\Gamma; T \;\; a\ \bot\ \varphi) \Rrightarrow (\Gamma'; T' \;\; a'\ e'\ \varphi'). \; \Gamma' \models (\lambda x.C[x])\ t \cong C[t] : T' \;\; a'\ e',$$

where $a.f = \bot$, *and* $\Gamma'(\varphi' \setminus x) <: a_f$.

Theorem 7.10 (β-Equivalence). *If* $\Gamma, x : T_1 \;\; a_1 \vdash t_2 : T_2 \;\; a\ e$, *and* $\Gamma \vdash t_1 : T_1 \;\; a_1\ \bot$, *and* $a_1.f = \bot$, *and* $\Gamma(\mathrm{fv}(t_2) \setminus x) <: a_f$, *then* $\Gamma \models (\lambda x.\ t_2)(t_1) \approx_{log} t_2[t_1/x] : T_2 \;\; a\ e$.

Theorem 7.11 (Reordering). *If* $\Gamma \vdash t_1 : \texttt{Bool} \;\; a_1\ e_1$, *and* $\Gamma \vdash t_2 : \texttt{Bool} \;\; a_2\ e_2$, *and* $e_1 \wedge e_2 = \texttt{false}$, *then* $\Gamma \models t_1 \otimes t_2 \approx_{log} t_2 \otimes t_1 : \texttt{Bool} \;\; \langle\bot\rangle\ e_1 \triangleright e_2$

8 Related Work

Purity In object-oriented programming languages, various definitions of purity have been proposed [Stewart et al. 2014], among which the definition of Leavens et al. [2006] has been widely adopted in Java. Under this definition, a location is modified when it exists in both the pre-state and post-state of an execution, but stores different values in these two states. This notion of observable purity has been adopted by Java analysis tools, including JPPA [Salcianu and Rinard 2005], ReIm and ReImInfer [Huang and Milanova 2012; Huang et al. 2012], JPure [Pearce 2011], as well as by Scala purity checking, *e.g.*, [Rytz et al. 2013].

Different from the above systems, Joe-E [Mettler et al. 2010] defines pure methods as being side effect free and deterministic, *i.e.*, their outputs should depend only on their inputs. This notation of purity is closer to the functional programming notion of purity.

In this paper, we use the definition of a pure function aligning with the mathematical definition of a function. That is, a pure function's behavior is entirely determined by its input argument, and its output does not depend on an external runtime environment that may change. This definition is suitable for our purpose of equational reasoning.

Sabry [1998] proposes to define purity for the Haskell language based on parameter-passing independence, *i.e.*, different evaluation strategies (call-by-value, call-by-need, or call-by-name) lead to equivalent values modulo divergence and runtime errors. Our treatment of termination behavior differs from Sabry's approach. His equivalence is defined modulo termination, so his definition of pure language describes Haskell more than Coq. In the age of Coq, Agda [Bove et al. 2009], Lean [de Moura and Ullrich 2021], and especially also Liquid Haskell [Vazou 2016], we take the position that it is useful to tighten the definition and recognize nontermination as impurity in closer correspondence with logical consistency. In the presence of possibly-infinite codata, a syntactic guardedness checker is used to ensure productivity, *i.e.*, even if a program generates an infinite amount of data, each piece will be generated in finite time [Turner 2004].

Type-and-Effect Systems Effect systems are introduced to integrate imperative operations into functional languages [Gifford and Lucassen 1986; Lucassen and Gifford 1988], to track read and write effects on the heap [Nielson et al. 1999], to check purity [Pearce 2011; Rompf et al. 2009; Rytz et al. 2013] and exceptions [Gosling et al. 2005; Leroy and Pessaux 2000; Pessaux and Leroy 1999], and others. [Marino and Millstein 2009] introduce a generic type-and-effect system that allow programmers to easily define new effects systems. Gordon [2021] introduce a generic effect quantale framework that can model the effects of a range of sequential effect systems. In this work, we adopt a canonical binary effect system that distinguishes between computations that may induce observable effects and those that do not, serving the purpose of reasoning about purity.

Algebraic Effects and Handlers Algebraic effects [Plotkin and Power 2003] and handlers [Plotkin and Pretnar 2013] are a purely functional composable approaches to modeling effects, and inherit similar challenges in supporting effect polymorphism as those encountered in traditional effect systems. These challenges are evident in languages that support algebraic effects, *e.g.*, Koka [Leijen 2017], Helium [Biernacki et al. 2020], the language in Hillerström et al. [2017]'s work, and the system developed by Zhang and Myers [2019]'s work. Frank [Convent et al. 2020; Lindley et al. 2017] reduces the annotation burden by leveraging ambient effects provided by the context, but relies on desugaring them to traditional effect polymorphism. This mechanism is fragile, and may result in unclear error messages. Tang and Lindley [2025]; Tang et al. [2025] extend Frank's approach with multimodal type theory [Gratzer 2023; Gratzer et al. 2021] in tracking modes for types and terms.

Capabilities 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], and to ensure unique access with borrowing [Haller and Odersky 2010], Reference capabilities in Pony [Clebsch 2017] distinguish read and write capabilities. Gordon [2020] articulates the use-mention distinction when designing capabilities and effect systems. Our λ_{ae}-calculus uses ability to track "mention", and effects to track "use". Craig et al. [2018] use capabilities to bound the effects that an expression can induce for a language with a fixed set of global resources. Our systems can reason about dynamically allocated resources.

Osvald et al. [2016] distinguish 1st- and 2nd-class values via privilege qualifiers, which can mediate access to different kinds of capabilities (*e.g.*, read and write) with a fine-grained privilege lattice. Effekt [Brachthäuser et al. 2020] treats effects as capabilities and treat all functions as second-class values, which providing a lightweight of effect polymorphism, but impeding reasoning about purity. Our systems track resource via abilities in a coarse-grained way and can recognize pure computation, which was not a goal of those prior systems.

Tracking Variables in Types Bao et al. [2021] introduce reachability types to track aliasing and separation through reachability qualifiers. Their work has been extended to support polymorphism [Wei et al. 2024], cyclic references [Deng et al. 2025] and bidirectional typing [Jia et al. 2024]. Our semantic model is adapted from Bao et al. [2025]'s work, but we focus on a simplified setting restricted to Boolean references. In addition, Bunting and Murawski [2025] study 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. In the future, we plan to extend our work to support fine-grained ability and effect tracking for programs with higher-order stores by leveraging reachability qualifiers as in Bao et al. [2025]'s work.

Capturing types [Boruch-Gruszecki et al. 2023; Xu et al. 2024] integrate capability tracking and escape checking into Scala 3. Treating effects as capabilities provides a lightweight way to achieve effect polymorphism [Brachthäuser et al. 2020]. Xu and Odersky [2024] introduce reach capabilities

to address the challenges when capture checking and mutable variables interact. Integrating mechanisms from our approach could potentially enable more precise purity checking for Scala.

9 Conclusions

This paper proposed a semantic framework for purity and a measure of expressiveness for effect typing disciplines. We showed that canonical binary effect and capability systems are incomparable, motivating a synthesis that combines their strengths, which we call *type, ability, and effect systems*. Our semantic definition of purity, along with our mechanized logical relation, provides a foundation for comparing and proving properties across diverse effect typing disciplines.

Acknowledgments

This paper has benefited from fruitful discussions at the 24th Meeting of IFIP WG 2.11 in Edinburgh, notably with Jeremy Gibbons, Sam Lindley, and others. This work was supported in part by NSF award 2348334 and an 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.

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

Yuyan Bao, Songlin Jia, Guannan Wei, Oliver Bračevac, and Tiark Rompf. 2025. Modeling Reachability Types with Logical Relations. *Proc. ACM Program. Lang.* 9, OOPSLA (2025).

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.

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

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29.

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.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *TPHOLs (Lecture Notes in Computer Science, Vol. 5674)*. Springer, 73–78.

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.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30.

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.

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

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9.

Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for Free. In *ICFEM (Lecture Notes in Computer Science, Vol. 11232)*. Springer, 231–247.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE (Lecture Notes in Computer Science, Vol. 12699)*. Springer, 625–635.

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).

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

Sophia Drossopoulou, Julian Mackay, Susan Eisenbach, and James Noble. 2025. Reasoning about External Calls. *arXiv preprint arXiv:2506.06544* (2025).

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

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.
Andrzej Filinski. 2010. Monads in action. In *POPL*. ACM, 483–494.
David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *LISP and Functional Programming*. ACM, 28–38.
Colin S. Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl). In *ECOOP (LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:25.
Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79.
James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2005. The Java(TM) Language Specification (3rd Edition).
Daniel Gratzer. 2023. *Syntax and semantics of modal type theory*. Ph. D. Dissertation. Aarhus University, Denmark.
Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Log. Methods Comput. Sci.* 17, 3 (2021).
Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *European Conference on Object-Oriented Programming*. Springer, 354–378.
Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs, Vol. 84)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19.
Wei Huang and Ana L. Milanova. 2012. ReImInfer: method purity inference for Java. In *SIGSOFT FSE*. ACM, 38.
Wei Huang, Ana L. Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: checking and inference of reference immutability and method purity. In *OOPSLA*. ACM, 879–896.
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]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Softw. Eng. Notes* 31, 3 (2006), 1–38.
Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499.
Xavier Leroy and François Pessaux. 2000. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.* 22, 2 (2000), 340–377.
Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.
John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57.
Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *TLDI*. ACM, 39–50.
Adrian Mettler, David A. Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *NDSS*. The Internet Society.
Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.
Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.
James H Morris. 1969. Lambda calculus models of programming languages. (1969).
Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. Type and effect systems. In *Principles of Program Analysis*. Springer, 283–363.
Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. 2021. Safer exceptions for Scala. In *SCALA/SPLASH*. ACM, 1–11.
Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. ACM, 234–251.
David J. Pearce. 2011. JPure: A Modular Purity System for Java. In *CC (Lecture Notes in Computer Science, Vol. 6601)*. Springer, 104–123.
François Pessaux and Xavier Leroy. 1999. Type-Based Analysis of Uncaught Exceptions. In *POPL*. ACM, 276–290.
Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.
Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94.
Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).
Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*. ACM, 317–328.
Lukas Rytz. 2014. *A Practical Effect System for Scala*. Ph. D. Dissertation. EPFL, Switzerland.
Lukas Rytz, Nada Amin, and Martin Odersky. 2013. A flow-insensitive, modular effect system for purity. In *FTfJP at ECOOP*. ACM, 4:1–4:7.
Amr Sabry. 1998. What is a Purely Functional Language? *J. Funct. Program.* 8, 1 (1998), 1–22.
Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *VMCAI (Lecture Notes in Computer Science, Vol. 3385)*. Springer, 199–215.

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

Arran D Stewart, Rachel Cardell-Oliver, and Rowan Davies. 2014. CSSE Technical Report UWA-CSSE-14-001 Side effect and purity checking in Java: a review. https://www.academia.edu/9622706/Side_effect_and_purity_checking_in_Java_a_review.

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (1994), 245–296.

Wenhao Tang and Sam Lindley. 2025. Rows and Capabilities as Modal Effects. arXiv:2507.10301 [cs.PL] https://arxiv.org/abs/2507.10301

Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1130–1157.

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.

Sam Tobin-Hochstadt. 2011. Is there an expressiveness hierarchy for type systems? Theoretical Computer Science Stack Exchange. https://cstheory.stackexchange.com/q/8238

D. A. Turner. 2004. Total Functional Programming. *J. Univers. Comput. Sci.* 10, 7 (2004), 751–768.

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover.* Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

Philip Wadler. 1992. The Essence of Functional Programming. In *POPL*. ACM Press, 1–14.

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Log.* 4, 1 (2003), 1–32.

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.

Guannan Wei, Oliver Bracevac, 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.

Anxhelo Xhebraj, Oliver Bracevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29.

Yichen Xu, Aleksander Boruch-Gruszecki, and Martin Odersky. 2024. Degrees of Separation: A Flexible Type System for Safe Concurrency. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1181–1207.

Yichen Xu and Martin Odersky. 2024. A Formal Foundation of Reach Capabilities. In *Programming*. ACM.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29.