Local Temporal Reasoning

Eric Koskinen * †
New York University

Tachio Terauchi ‡
Japan Advanced Institute of Science and Technology

Abstract

We present the first method for reasoning about temporal logic properties of higher-order, infinite-data programs. By distinguishing between the finite traces and infinite traces in the specification, we obtain rules that permit us to reason about the temporal behavior of program parts via a type-and-effect system, which is then able to compose these facts together to prove the overall target property of the program. The type system alone is strong enough to derive many temporal safety properties using refinement types and temporal effects. We also show how existing techniques can be used as oracles to provide liveness information (*e.g.* termination) about program parts and that the type-and-effect system can combine this information with temporal safety information to derive nontrivial temporal properties. Our work has application toward verification of higher-order software, as well as modular strategies for procedural programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; Correctness proofs; Reliability; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Languages, Theory, Verification

Keywords Higher-order programs, formal verification, temporal logic, program analysis, model checking

1. Introduction

Programming languages that use higher-order functionality (*e.g.* Java, C#, F#, Haskell, Ocaml, Perl, Python, Ruby) have become commonplace. Higher-order language features such as `map`, `grep`, Google's Map/Reduce, are used widely and applauded for their simplicity and modularity.

Meanwhile, in the past couple of decades, algorithms and tools have emerged that have enabled automatic verification of some industrial software systems. Symbolic analysis techniques such as abstraction refinement [7] and interpolation [23] have given rise to interprocedural program analysis tools for safety [2, 5], while termination argument refinement [24] has led to tools for liveness [8, 13]. Further research has led to tools and algorithms for verifying properties expressed in temporal logic: more elaborate specifications that combine safety and liveness [1, 4, 9–12].

The verification techniques discussed thus far have been mostly limited to imperative *first*-order software and cannot be applied to *higher*-order programming languages. In recent years, researchers have developed some techniques for verifying higher-order programs. Some showed how to verifying temporal properties of higher-order programs when the domain of data is finite (*i.e.* boolean). Others have also developed methods of verifying purely safety properties [16, 19, 25, 28] or purely termination [20, 21] of higher-order programs with infinite data (*e.g.* integers) . Despite the efforts discussed above, at present there are no methods for verifying safety/liveness properties (*i.e.* temporal logic formulae) of programs written in higher-order languages.

We present the first technique for verifying temporal logic properties of higher-order, infinite-data programs. The crux of our work is to decompose the problem, not only by dividing the program up into individual expressions via a type-and-effect system, but also, for every expression, to track the behavior of *finite* traces separate from the behavior of *infinite* traces. Our type rules permit verification oracles (including the type system itself) to reason about the conditional safety and liveness (*i.e.* temporal) behavior of program parts, and compose these facts together to prove the overall target property of the program. Moreover, we show that existing tools can be used as oracles to introduce liveness proofs into the type system's effects. While it is a commonly held belief that type systems cannot be used for liveness properties, we show how they can, nonetheless, be used to *carry* some liveness information and soundly *combine* reasoning about program parts together to prove overall safety and liveness.

By way of an example, consider a function application $e\ v$, where we are attempting to prove an overall temporal property "**tick** $\cup$ **boom**," which means that some event **tick** will occur repeatedly until the event **boom** occurs, and **boom** is inevitable. With the type-and-effect system described in this paper we can, for example, reason about the *safety* behavior

$$\Gamma \vdash e : \tau \xrightarrow{\textbf{boom}} \tau' \ \&\ \textbf{tick}^*$$

that is due to the reduction of e (repeating event **tick**) and the latent behavior (event **boom**) that arises when e is applied to a value $v : \tau$, and then separately reason about the *termination* of (the reduction of) e via an oracle:

$$\Gamma \vdash e\ v : \tau' \ \&\ \text{terminates}$$

---

* Supported in part by the CMACS NSF Expeditions in Computing award 0926166.

† Supported in part by the Japan Society for the Promotion of Science (JSPS).

‡ Supported in part by MEXT Kakenhi 26330082 and 25280023.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM 978-1-4503-2886-9…$15.00.
http://dx.doi.org/10.1145/2603088.2603138

We are using shorthand in these judgments' effects, whereas formally our type system distinguishes the behavior of *finite* traces from the behavior of *infinite* traces. The type-and-effect system in this paper combines all of this information together via refinement types, and temporal connectives (intersection, union, concatenation) to obtain an overall goal judgment

$$\Gamma \vdash e\ v : \tau' \,\&\, \underline{\textbf{tick}} \cup \underline{\textbf{boom}}$$

while carefully accounting for possibly divergent evaluations. Our formalism permits oracles of arbitrary temporal expressive power. Our use of termination in the example above is a special case.

Contributions. To our knowledge, this work marks the first method for reasoning about temporal logic properties of higher-order programs that have infinite data. The above is a limited example. We believe our work provides the theoretical foundation toward several areas of practical significance. To this end, we have devised general rules so there are many instantiations and applications of them, including:

1. Instantiation to a wide variety of specification logics. We are able to support any logic that is closed under intersection, union, and composition (over finite and infinite traces) such as Büchi specifications. We sometimes use LTL as a shorthand for Büchi [15].
2. Instantiation to arbitrary type environments. Often the type system alone is strong enough to derive safety properties. For example, when using refinement types in the absence of a termination oracle, our rules can be thought of as a novel extension to dependent types, where temporal behaviors are carried as effects.
3. Instantiation of oracles to any fragment of program expressions or any subset of the specification logic.
4. Instantiation to a modular reasoning system for temporal behaviors of *first*-order procedural programs.

We have devised our methodology to be based on *local* reasoning, employing a type system. This stands in contrast to many existing verification works for higher-order programs that operate by extracting a transition system and then performing standard model checking techniques. Such existing techniques suffer from the inability to refine the abstraction during the verification process and, moreover, require input programs to be given in CPS form.

Limitations. It is a commonly held belief that (ordinary) type systems cannot be used to derive liveness properties. In this paper, we do not allow the type system to derive (non-trivial) liveness properties by itself. However, when liveness information is introduced by an oracle (*e.g.* a termination oracle), our type system is able to soundly combine this liveness information with safety/liveness information from other oracles or the type system itself. To maintain soundness, we also have to be careful when typing recursive functions, as we will discuss in the next section. Finally, note that the temporal behavior of a program is directly related to its evaluation order. For the purposes of this paper, we assume a strict evaluation order.

Organization. We first give a high-level description of our methodology in Section 2. After preliminaries in Section 3, we present our type and effect system in Section 4. We prove type soundness (Theorem 4.1). We then examine a variety of examples, discussed in Section 5, that span a range of temporal behaviors, compositions and oracle power. We conclude with a discussion of related work in Section 6.

2. Overview

Consider the following higher-order program, written in an ML-like syntax:

Example 1.

```
let rec zoom _ = ev[zoom]; zoom ()
and shrink f = ev[shrink];
      if ( f () = 0 ) then
        zoom ()
      else
        shrink (λ_. (f ()) - 1)
and main() = ev[main];
      let t = *pos in
        shrink (λ_. t)
```

For the moment, think of the boxed expressions as skip. The function zoom calls it self recursively, looping forever. Next, shrink is a recursive function and it takes, as an argument, a function f from `unit` to `int`. If applying f returns 0, zoom is called. Otherwise, shrink is called recursively with a new partially-defined argument. Finally, main initiates with the expression shrink $(\lambda\_.\ \mathsf{t})$ where t is bound to a nondeterministically chosen positive integer (denoted $*^{\mathsf{pos}}$).

For this example, we may want to prove the Linear Temporal Logic (LTL) property: $\Phi = \underline{\textbf{main}} \wedge \mathsf{X}(\underline{\textbf{shrink}}\ \mathsf{U}\ \underline{\textbf{zoom}})$. (We are abusing notation here and using LTL as shorthand for a Büchi specification over finite *and* infinite executions, but we will discuss that shortly.) This property consists of the temporal operators X and U as well as atomic propositions that are *events*, denoted in the event **font**. Events are emitted by the $ev[\underline{\textbf{event}}]$ expression. For simplicity, in the above example we correlate events with function calls, indicating that the function with the corresponding name has been called (this correlation is also a suitable definition of events in the context of practical examples). The property Φ specifies that the main function will be called first (event **main**) and then in the next step, the shrink function will be called (event **shrink**) repeatedly Until zoom is called (event **zoom**). U is the *strong* until operator, which specifies that its second argument must hold after finitely many steps.

Intuitively, we know that Φ holds of the above example. The property Φ is a trace-based property and specifies that every (terminating or nonterminating) execution of this program must generate event sequences of the form:

$$\underline{\textbf{main}}, (\underline{\textbf{shrink}})^*, \underline{\textbf{zoom}}, \ldots$$

where there are *finitely* many instances of **shrink**. The property holds because, when shrink is called with a function that returns a positive integer, shrink will eventually call zoom.

Proving that this property holds of this program is challenging. There are several obstacles: First it consists of both safety and liveness aspects, in the form of reachability and termination. Second, there is nondeterministic input. Third, the state space is infinite. Finally, this is a higher-order program. There are several recent related works, but all of them have restrictions that make them unsuitable to this example. Some works are restricted to finite data [18], others can reason over infinite data but are restricted to safety [28] or termination [20] properties, and still others can reason about infinite-state systems but are restricted to first-order input programs [10].

Compositional reasoning. The key idea of this paper is to decompose the problem of verifying that the entire program satisfies Φ into pieces in two ways. The first decomposition is not surprising: we divide the program up into its individual expressions so that we can determine which events arise from a given expression. For example, the behavior of a function application $e\ v$ is determined by separately considering the behavior of the value v as it is reduced, the behavior of the function expression e as it is reduced and, finally, the latent behavior that arises when e is applied to v.

The second form of decomposition is that, for every expression, we track the event behavior of *finite* traces separate from the event behavior of *infinite* traces. That is, for an expression e we will have two specifications: Ξ for the finite part and Π for the infinite part. The event behavior of e is the union of these specifications. Büchi automata provide a convenient such specification language that can be characterized separately over finite and infinite runs. (Büchi automata are also known to be closed under concatenation across the two partitions [29], which we will see to be important in Section 3.) The informed reader will note two commonly held beliefs. First, liveness is generally not composable: one cannot, in general, find a witness to non-liveness with a finite prefix of a trace. Related is the second belief that per-expression reasoning, as seen in type systems, cannot be used for liveness. However, as we will see here, type systems can be used to *carry* temporal safety and liveness specifications for expressions. This fact, combined with our separation of finite traces from infinite traces, allows us to soundly *combine* reasoning about program parts together to prove overall safety and liveness.

Our technique is best illustrated through the running example. After the **main** event fires and t is bound, we need to show that the body of main, shrink ($\lambda\_.$ t), satisfies **shrink** U **zoom**. In this paper, we characterize the temporal behavior of program expressions via a type-and-effect system that distinguishes between finite and infinite event traces. Syntactically, judgments are of the form:

$$\Gamma \vdash e : \tau \,\&\, (\Xi, \Pi)$$

which denotes that, under a typing context Γ, we deduce that expression e has a (refinement) type τ, that finite event traces of e satisfy Ξ, and that infinite event traces of e satisfy Π.

Our type system permits us to decompose our reasoning about Example 1 into separate components:

1. *The latent temporal behavior arising from* shrink *when it is applied to an argument*:

$$\boxed{J_1}\quad \Gamma \vdash \mathsf{shrink} : (\mathtt{unit} \to \mathtt{int}) \xrightarrow{\mathbf{shrink}\,\mathsf{W}\,\mathbf{zoom}} \mathtt{unit} \,\&\, \varepsilon$$

   The above judgment J_1 says that shrink is a function which consumes a function and returns unit (*i.e.*, "()"). shrink itself is a function *value* and, as such, it generates no effects, denoted ε. However, when shrink is applied, the latent behavior "**shrink** W **zoom**" occurs, indicating that event **shrink** occurs until event **zoom** (but not that **zoom** will necessarily inevitably occur). Formally, as we will discuss later in the paper, the latent effects are also represented as a pair (Ξ, Π) of finite and infinite traces (*e.g.*, as finite and Büchi automaton). But, we are abbreviating it in LTL by encoding terminating runs as self-loops, as is commonly done.

2. *The conditions under which* shrink *terminates*:

$$\boxed{J_2}\quad \Gamma \vdash \mathsf{shrink} : (\mathtt{unit} \to \{i \mid i \geq 0\}) \xrightarrow{\mathsf{F}\neg\mathbf{shrink}} \mathtt{unit} \,\&\, \varepsilon$$

   We use refinement (dependent) types [31] to say that the argument of shrink is a function that returns an integer above 0. In this judgment J_2, the latent effect is F¬**shrink**, indicating that eventually an event will occur that is not **shrink**. For convenience, we will sometimes abbreviate such termination judgments as "$\Gamma \vdash e : \tau$ & terminates".

   Notice that in the above judgment we have used a refinement type system to place the *conditions* on (the function passed to) shrink under which shrink will terminate: that the passed function is returning a nonnegative number.

Our type-and-effect system combines these facts (whose origins we will next discuss) together to obtain a final judgment, using a few rules such as function application (**App**), the combination rule (**Comb**), and the subtyping rule (**Sub**):

$$\dfrac{\dfrac{\dfrac{\boxed{J_1}\quad \ldots\,(\lambda\_.\ \mathsf{t})\,\ldots}{\vdots}\,\mathbf{App} \qquad \dfrac{\boxed{J_2}\quad \ldots\,(\lambda\_.\ \mathsf{t})\,\ldots}{\vdots}\,\mathbf{App}}{\vdots}\,\mathbf{Comb}}{\Gamma, \mathsf{t} : \{i \mid i \geq 0\} \vdash \mathsf{shrink}\ (\lambda\_.\ \mathsf{t}) : \mathtt{unit} \,\&\, \mathbf{shrink}\ \mathsf{U}\ \mathbf{zoom}}\,\mathbf{Sub}$$

This final judgment indicates that the expression (shrink $\lambda\_.$ t) has unit type and will exhibit a finite sequence of **shrink** events, followed by a **zoom** event.

Our type-and-effect system, formalized in Section 4, consists of several other rules that combine temporal information about expressions, taking care to account for possibly divergent computation. To be able to combine rich information across different oracles, we adopt and extend the refinement type system that has garnered popularity in the verification of functional programs [6, 16, 19, 25, 28, 30]. There are typing rules for all the usual higher-order features such as subtyping, intersection, etc—however, they will now also carry the temporal effect of each (sub)expression.

In the above example, our work combined a derivation J_1, which is a *safety* proof with a derivation J_2 which is a *termination* proof. But one is inclined to wonder: where do these pieces come from in the first place? We will now describe how these subproofs are obtained, respectively.

Safety via the type system. We can use our type-and-effect system to deduce that the function shrink has the latent (safety) effect **shrink** W **zoom**. This arises as a fixpoint solution to the typing context in the judgments over the body of shrink. Assume that we have a typing environment Γ that already contains a judgment:

$$\Gamma(\mathsf{zoom}) = \mathtt{unit} \xrightarrow{\mathsf{G}\ \mathbf{zoom}} \mathtt{unit} \,\&\, \varepsilon$$

indicating that when zoom is applied to unit an arbitrarily long sequence of **zoom** events may occur. We also know that an application of shrink will generate event **shrink**. (We assume that every named function begins with a special event statement of the same name, but this has been omitted for readability. For example, the body of shrink in Example 1 is "$ev[$**shrink**$]$; **if** ...").

Depending on which branch is taken in shrink, either shrink will recur, generating event **shrink** or else the expression zoom () will generate event G **zoom**. Since we do not know *a priori* whether f () = 0 returns true or false, a valid typing context will say that shrink f satisfies **shrink** ∨ G **zoom**. This disjunction of event atomic propositions is valid, however, it only asserts the event currently generated. If we look for a fixpoint of

$$\alpha \;=\; \mathbf{shrink} \,\wedge\, \mathsf{X}(\mathsf{G}\ \mathbf{zoom} \,\vee\, \alpha)$$

there is a stronger solution:

$$\Gamma, \mathsf{zoom} : \ldots \vdash \mathsf{shrink}\ \mathsf{f} : \mathtt{unit} \,\&\, \mathbf{shrink}\ \mathsf{W}\ (\mathsf{G}\ \mathbf{zoom})$$

Note that type systems typically do not distinguish fixpoints (*e.g.* greatest versus least) and, as such, we cannot conclude anything about the infinite traces. However, we can combine a judgement of **shrink** W (G **zoom**) over the finite traces, with an oracle judgement that G(**shrink** ∨ **zoom**) holds over the infinite traces, to obtain that **shrink** W (G **zoom**) holds over all traces[1].

Liveness via a termination oracle. Type systems themselves cannot generate liveness information. However, we show how they

[1] We conjecture that there is a way to organize type systems so that infinite-trace behaviors of expressions can arise from fixpoint equations, but we leave this to future work.

$$
\begin{array}{lcl}
P & ::= & P \cup \{\mathtt{F}\ \bar{x} = e\} \mid \varnothing \\
\underline{\mathbf{a}} & ::\in & \Sigma \\
e & ::= & x \mid c \mid \mathtt{F} \mid \mathtt{ev}[\underline{\mathbf{a}}] \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid x\,y \\
 & \mid & x\ op\ y \mid \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \lambda x.e
\end{array}
$$

Figure 1: The syntax of the simple functional language.

can be used to carry liveness (or safety) information obtained from an outside source. This is accomplished via an *oracle* rule:

$$\frac{\Theta \blacktriangleright e : \tau \,\&\, \Phi}{\Delta, \Theta \vdash e : \tau \,\&\, \Phi}\ \textbf{Oracle}$$

(Note that an oracle $\blacktriangleright$ must satisfy the Oracle property in Definition 4.1.) This rule is very general and it allows us to incorporate information from external techniques into our type-and-effect system. Let us look again at the body of shrink. If f is a function that, when applied to (), returns 0, then zoom will be invoked. Otherwise, shrink is called recursively with a partial function definition $(\lambda\_.\ (\mathsf{f}\,())-1)$. Intuitively we know that this recursive function terminates provided that it is applied to a function f that, when applied to (), returns a positive integer. In recent work [20] we describe a technique for proving termination of higher-order programs. We can adapt this technique to prove a temporal property ($\mathsf{F}\neg\underline{\mathbf{shrink}}$) that is similar to termination, under the *condition* that f() is non-negative. With this proof in hand, we can use the oracle rule to conclude that

$$\Gamma \vdash \mathsf{shrink} : (\mathtt{unit} \to \{i \mid i \geq 0\}) \xrightarrow{\mathsf{F}\neg\underline{\mathbf{shrink}}} \mathtt{unit} \,\&\, \varepsilon$$

Of course in general, the oracle rule can be used to introduce more elaborate temporal specifications. We will see such examples in Section 5.

The example in this section provided a small illustration of how our work combines temporal reasoning over separate program pieces together to prove overall temporal properties. In Section 5 we present a variety of examples that illustrate many ways that our work can be instantiated. In addition to verification of higher-order programs, our work also suggests a local technique for verification of first-order interprocedural programs that would escape previous restrictions on the logic [1]. We now turn to a formal presentation of our work.

3. Preliminaries

We focus on a small functional language shown in Figure 1. A *program*, P, is a finite set of mutually recursive function definitions. $\mathtt{F}\,\bar{x} = e$ uniquely defines a function named $\mathtt{F}$ with the formal parameters $\bar{x}$ and the closed expression e as the body. Function names may appear free in a function body. The notation $\bar{x}$ denotes a possibly empty sequence. Note that nested recursive function definitions can be supported via λ lifting [17].

Expressions, e, comprises constants c, function names $\mathtt{F}$, variables x, lambda abstractions $\lambda x.e$, constant operations $x\ op\ y$, function applications $x\,y$, event raises $\mathtt{ev}[\underline{\mathbf{a}}]$, conditional branches $\mathtt{if}\ x\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$, and let expressions $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$. In $\mathtt{ev}[\underline{\mathbf{a}}]$, $\underline{\mathbf{a}} \in \Sigma$ is an event symbol. We assume that the set of event symbols Σ contains a special event symbol $\underline{\mathbf{step}}$ whose role is explained below. The constants include the boolean constants true and false, the unit constant $()$, and the integer constants. The operator *op* include boolean and integer operators such as $+$, $-$, $\leq$, as well as the operator $*_{\mathtt{int}}$ that returns a non-deterministic integer (for simplicity, we often write $*_{\mathtt{int}}$ for $\_\ *_{\mathtt{int}}\ \_$). We write $\mathit{fv}(e)$ for the free variables of e.

$$\frac{}{\Gamma \vdash^s c : \mathit{sty}(c)}$$

$$\frac{\Gamma \vdash^s x : B_1 \qquad \Gamma \vdash^s y : B_2 \qquad \mathit{sty}(op) = B_1 \to B_2 \to B_3}{\Gamma \vdash^s x\ op\ y : B_3}$$

$$\frac{}{\Gamma \vdash^s \mathtt{F} : \Gamma(\mathtt{F})} \quad \frac{}{\Gamma \vdash^s x : \Gamma(x)}$$

$$\frac{\Gamma, x{:}s \vdash^s e : s'}{\Gamma \vdash^s \lambda x.e : s \to s'} \quad \frac{}{\Gamma \vdash^s \mathtt{ev}[\underline{\mathbf{a}}] : \mathtt{unit}}$$

$$\frac{\Gamma \vdash^s e_1 : s \quad \Gamma, x{:}s \vdash^s e_2 : s'}{\Gamma \vdash^s \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : s'}$$

$$\frac{\Gamma \vdash^s x : s \to s' \quad \Gamma \vdash^s y : s}{\Gamma \vdash^s x\,y : s'}$$

$$\frac{\Gamma \vdash^s x : \mathtt{bool} \quad \Gamma \vdash^s e_1 : s \quad \Gamma \vdash^s e_2 : s}{\Gamma \vdash^s \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : s}$$

$$\frac{\forall \mathtt{F}\ \bar{x} = e \in P. \Gamma \vdash^s \lambda \bar{x}.e : \Gamma(\mathtt{F})}{\Gamma \vdash^s \star}$$

Figure 2: Simple type system type checking rules.

For simplicity, we restrict branch conditions, operator arguments and the expressions in function applications to variables. But note that a non-variable form can be encoded by using let (*e.g.* $e_1\,e_2$ can be encoded as $\mathtt{let}\ x = e_1\ \mathtt{in}\ \mathtt{let}\ y = e_2\ \mathtt{in}\ x\,y$). For convenience, we use the non-variable forms when the encoding is clear from the context.

As usual, applications associate to the left so that $e_0\,e_1\,e_2 = (e_0\,e_1)\,e_2$. We write $e_1; e_2$ for $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ such that $x \notin \mathit{fv}(e_2)$. Without loss of generality, we assume that bound variables are distinct. We say that an expression is closed if it contains no free variables.

We assume that the program is simply typed. For an expression e in the program, write $\mathit{sty}(e)$ for its simple type. A simple type, s, is defined by the following grammar.

$$
\begin{array}{lcl}
B & ::= & \mathtt{int} \mid \mathtt{bool} \mid \mathtt{unit} \\
s & ::= & B \mid s_1 \to s_2
\end{array}
$$

Figure 2 defines the simple typing judgement $\vdash^s$. A program P is simply typed if $\Gamma \vdash^s \star$. (The program is implicit and is omitted in the type judgement rules.)

3.1 Semantics

We define the call-by-value semantics of the language via big-step evaluation rules. We define terminating evaluation by the usual inductive rules ($\Downarrow_P$), and define non-terminating runs by co-inductive rules ($\Uparrow_P$), following the previous work on co-inductive big-step [14, 22]. Non-terminating runs need be provided as well as the usual terminating ones because we are concerned with possibly infinite sequences of events generated by the program.

The terminating judgement $e \Downarrow_P v \,\&\, \varpi$ expresses that the expression e evaluates to the value v producing the finite sequence of events $\varpi \in \Sigma^*$. A *value* is either a constant or a closed $\lambda x.e$ (i.e., $\mathit{fv}(e) \subseteq \{x\}$). The non-terminating judgement $e \Uparrow_P \bot \,\&\, \pi$ expresses that e diverges and produces the infinite sequence of events $\pi \in \Sigma^\omega$. Because of non-determinism, an expression can have multiple terminating and non-terminating evaluations. We

TERMINATING RUNS

$$\frac{e[v/x] \Downarrow_P v' \,\&\, \varpi}{(\lambda x.e)\, v \Downarrow_P v' \,\&\, \underline{\mathbf{step}} \cdot \varpi}\ \textbf{st-App}$$

$$\frac{[\![op]\!](c_1, c_2) = c}{c_1\ op\ c_2 \Downarrow_P c \,\&\, \varepsilon}\ \textbf{st-Op} \qquad \frac{}{v \Downarrow_P v \,\&\, \varepsilon}\ \textbf{st-Val}$$

$$\frac{\mathtt{F}\ \bar{x} = e \in P}{\mathtt{F} \Downarrow_P \lambda \bar{x}.e \,\&\, \varepsilon}\ \textbf{st-Fun} \qquad \frac{}{\mathtt{ev}[\underline{\mathbf{a}}] \Downarrow_P () \,\&\, \underline{\mathbf{a}}}\ \textbf{st-Ev}$$

$$\frac{e_1 \Downarrow_P v \,\&\, \varpi_1 \quad e_2[v/x] \Downarrow_P v' \,\&\, \varpi_2}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow_P v' \,\&\, \varpi_1 \cdot \varpi_2}\ \textbf{st-Let}$$

$$\frac{e_1 \Downarrow_P v \,\&\, \varpi}{\mathtt{if}\ \mathsf{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow_P v \,\&\, \varpi}\ \textbf{st-If1}$$

$$\frac{e_2 \Downarrow_P v \,\&\, \varpi}{\mathtt{if}\ \mathsf{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow_P v \,\&\, \varpi}\ \textbf{st-If2}$$

NON-TERMINATING RUNS

$$\frac{e[v/x] \Uparrow_P \bot \,\&\, \pi}{(\lambda x.e)\, v \Uparrow_P \bot \,\&\, \underline{\mathbf{step}} \cdot \pi}\ \textbf{snt-App}$$

$$\frac{e_1 \Uparrow_P \bot \,\&\, \pi}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Uparrow_P \bot \,\&\, \pi}\ \textbf{snt-Let1}$$

$$\frac{e_1 \Downarrow_P v, \varpi \quad e_2[v/x] \Uparrow_P \bot \,\&\, \pi}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Uparrow_P \bot \,\&\, \varpi \cdot \pi}\ \textbf{snt-Let2}$$

$$\frac{e_1 \Uparrow_P \bot \,\&\, \pi}{\mathtt{if}\ \mathsf{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Uparrow_P \bot \,\&\, \pi}\ \textbf{snt-If1}$$

$$\frac{e_2 \Uparrow_P \bot \,\&\, \pi}{\mathtt{if}\ \mathsf{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Uparrow_P \bot \,\&\, \pi}\ \textbf{snt-If2}$$

Figure 3: Semantics of the simple functional language

denote concatenation of finite event sequences ϖ_1 and ϖ_2 by $\varpi_1 \cdot \varpi_2$, and the concatenation of an finite event sequence ϖ to an infinite event sequence π by $\varpi \cdot \pi$.

Figure 3 shows the evaluation rules. We describe the key rules. The rule **st-Ev** evaluates $\mathtt{ev}[\underline{\mathbf{a}}]$ to a unit value, producing the event $\underline{\mathbf{a}}$. The rule **st-App** evaluates a function application $(\lambda x.e)\, v$. The rule handles the case where the function call (i.e,. the evaluation of $e[v/x]$) terminates. The special event $\underline{\mathbf{step}}$ is used to prevent hidden computation. The rule **snt-App** is similar to **st-App**, but for the case the call diverges. In **st-Op**, $[\![op]\!]$ denotes the semantics of the operator op. For example, $[\![+]\!](i, j) = i + j$ for any $i, j \in \mathbb{Z}$, and $[\![*_{\mathtt{int}}]\!](i, j) = k$ for any $i, j, k \in \mathbb{Z}$.

Remark: Hidden Computation. In temporal verification of first-order as well as higher-order programs there is some degree of engineering choice that needs to be made as to what constitutes a subsequent state. Related is the choice of the atomic proposition language and what aspect of the state are represented by that language. Imagine, for example, the imperative program: "`x:=1; increment(x); x:=-1;`" The property $\mathsf{F}(\mathtt{x} < 0)$ holds unless, of course, `increment(x)` diverges. It depends greatly on what is defined to be (*i*) the "state" of the program automaton and (*ii*) the process of computing the subsequent such state. Trouble arises when it is possible for *ii* to execute infinitely many steps.

We think that this is an important point and so, in this paper, we have given a semantics for the λ-calculus that ensures that there are no infinite, invisible computations. We do this by inserting $\underline{\mathbf{step}}$ events, as described above. We argue that future temporal verification techniques and tools for both first- and higher-order should be developed with this in mind. In the context of first-order programs, a state is typically defined to be the valuation of all program variables. This precludes the observation that there can be infinite computation over registers. We urge future semantics for first-order programs to incorporate a strategy similar to ours in order to ensure there is no hidden computation. There is, however, a caveat: user specifications may be impacted. A specification such as $\mathsf{F}p$ holds regardless of how many step events occur before p. A specification such as $\mathsf{G}p$ must be written with the possibility of $\underline{\mathbf{step}}$ events in mind. In most cases, the intention of a $\mathsf{G}p$ specification is really $\mathsf{G}(\underline{\mathbf{step}} \vee p)$. □

Returning to the language semantics, the rule **snt-App** is also used to evaluate function applications:

$$\frac{e[v/x] \Uparrow_P \bot \,\&\, \pi}{(\lambda x.e)\, v \Uparrow_P \bot \,\&\, \underline{\mathbf{step}} \cdot \pi}\ \textbf{snt-App}$$

This rule handles the case when the function call diverges (i.e., the evaluation of $e[v/x]$ diverges).

Example 2. Let P be the following four-function program:

$$\begin{array}{l} \mathtt{F}\ g = \mathtt{ev}[\underline{\mathbf{a}}]; \mathtt{if}\ *_{\mathtt{int}}\ \mathtt{then}\ g\, 0\ \mathtt{else}\ \mathtt{F}\,(\mathtt{H}\, g) \\ \mathtt{G}\ x = \mathtt{ev}[\underline{\mathbf{b}}]; 1 \\ \mathtt{H}\ g\ x = \mathtt{ev}[\underline{\mathbf{c}}]; g\, x \\ \mathtt{I}\ () = \mathtt{F}\,\mathtt{G}; \mathtt{F}\,\mathtt{G} \end{array}$$

Let $\hat{a} = \underline{\mathbf{step}} \cdot \underline{\mathbf{a}}$, $\hat{b} = \underline{\mathbf{step}} \cdot \underline{\mathbf{b}}$, and $\hat{c} = \underline{\mathbf{step}} \cdot \underline{\mathbf{c}}$. Let A be the regular language $\hat{a}^* \cdot \hat{c}^* \cdot \hat{b}$. Then, $\mathtt{I}\ () \Downarrow_P v, \varpi$ if and only if $\varpi \in \Xi$ and $v = 1$, and $\mathtt{I}\ () \Uparrow_P \bot, \pi$ if and only if $\pi \in \Pi$ where

$$\begin{array}{rcl} \Xi & = & \{\varpi \cdot \varpi' \mid \varpi \in A \cup \{\varepsilon\} \wedge \varpi' \in A\} \\ \Pi & = & \{\bot, \varpi \cdot \hat{\underline{\mathbf{a}}}^\omega \mid \varpi \in A \cup \{\varepsilon\}\} \end{array}$$

We often omit the program P from the definitions when the program is clear from the context. For example, we simply write $\Downarrow$ and $\Uparrow$ instead of $\Downarrow_P$ and $\Uparrow_P$.

4. Type System

We now formalize the type and effect system. Intuitively, our type and effect system is an *orchestrating language* that glues together information from the temporal property verifier oracles (termination checker, safety checker, etc.). As we described in Section 2, in order to be able to communicate rich information across the different oracles, we adopt and extend the refinement (dependent) type system that has garnered recent popularity [6, 16, 19, 25, 28, 30].

Figure 4 defines the syntax of types. Per previous work on refinement type systems, types include *refinement base types* $\{u{:}B \mid \theta\}$ that refines the base-type B by the refinement predicate θ which is a formula in some first-order logic theory on base-type

$$\begin{array}{rcl} \Xi & ::\subseteq & \Sigma^* \\ \Pi & ::\subseteq & \Sigma^\omega \\ \Phi & ::= & (\Xi, \Pi) \\ B & ::= & \mathtt{int} \mid \mathtt{bool} \mid \mathtt{unit} \\ \tau, \sigma & ::= & \{u{:}B \mid \theta\} \mid x{:}\sigma \xrightarrow{\Phi} \tau \end{array}$$

Figure 4: Type syntax

$$
\begin{array}{rcl}
(\Xi_1, \Pi_1) \cup (\Xi_2, \Pi_2) & = & (\Xi_1 \cup \Xi_2, \Pi_1 \cup \Pi_2) \\
(\Xi_1, \Pi_1) \cap (\Xi_2, \Pi_2) & = & (\Xi_1 \cap \Xi_2, \Pi_1 \cap \Pi_2) \\
(\Xi_1, \Pi_1) \cdot (\Xi_2, \Pi_2) & = & (\Xi_1 \cdot \Xi_2, \Pi_1 \cup (\Xi_1 \cdot \Pi_2)) \\
(\Xi_1, \Pi_1) \subseteq (\Xi_2, \Pi_2) & \Leftrightarrow & \Xi_1 \subseteq \Xi_2 \wedge \Pi_1 \subseteq \Pi_2
\end{array}
$$

Figure 5: Trace set operations.

$$
\frac{u \notin \mathit{fv}(\llbracket\Gamma\rrbracket_{base}) \quad \models \llbracket\Gamma\rrbracket_{base} \wedge \theta_1 \Rightarrow \theta_2}{\Gamma \vdash \{u{:}B \mid \theta_1\} \le \{u{:}B \mid \theta_2\}}
$$

$$
\frac{\Gamma \vdash \sigma_2 \le \sigma_1 \quad \Gamma, x : \sigma_2 \vdash \tau_1 \le \tau_2 \quad \Phi_1 \subseteq \Phi_2}{\Gamma \vdash x{:}\sigma_1 \xrightarrow{\Phi_1} \tau_1 \le x{:}\sigma_2 \xrightarrow{\Phi_2} \tau_2}
$$

Figure 8: Subtyping rules.

variables. We sometimes abbreviate $\{u{:}B \mid \theta\}$ as B when θ is a tautology (*e.g.*, $\{u{:}B \mid \top\} = B$). Intuitively, $\{u{:}B \mid \theta\}$ denotes the type of some value u of the base type B satisfying the formula θ.

The type $x : \sigma \xrightarrow{\Phi} \tau$ is a *dependent function type*, consisting of the argument type σ and the return type τ and the *latent effect* Φ. We extend the dependent function type from previous work by adding trace sets as a latent effect. Intuitively, $x : \sigma \xrightarrow{\Phi} \tau$ denotes the type of a function that returns a value of the type $\tau[y/x]$ and generates the events represented by the effect Φ when applied to an argument y of the type σ. As usual, $\rightarrow$ associates to the right. The system is parametrized by the class of trace sets that are usable as the effects.

An effect is a pair comprising a set of finite traces Ξ (the *terminating part*) and a set of infinite traces Π (the *non-terminating part*). We define operations on effects component-wise as shown in Figure 5. Here, $\Xi_1 \cdot \Xi_2$ denotes the concatenation of the traces in Ξ_1 to those in Ξ_2 and is defined $\{\varpi_1 \cdot \varpi_2 \mid \varpi_1 \in \Xi_1 \wedge \varpi_2 \in \Xi_2\}$. Similarly, a concatenation of a set of finite traces Ξ to a set of infinite traces Π is defined as $\Xi \cdot \Pi = \{\varpi \cdot \pi \mid \varpi \in \Xi \wedge \pi \in \Pi\}$.

The type system may be used by instantiating the effects with any class of trace sets that are closed under the operations. In the examples in this paper, we use a pair of finite state automaton and Büchi automaton (*i.e.*, regular and ω-regular trace sets) which have the required closure property [29]. We use LTL as a shorthand for trace sets by interpreting a finite trace as the infinite trace that self-loops at the last event (*e.g.*, $\perp = \emptyset$, $\top = \Sigma^*$, and $\underline{\mathbf{a}} \mathbin{\mathsf{U}} \underline{\mathbf{b}} = \underline{\mathbf{a}}^* \cdot \underline{\mathbf{b}} \cdot \Sigma^*$, for finite traces).

The type $\{x{:}B \mid \theta\}$ binds x in θ. Likewise, $x : \sigma \xrightarrow{\Phi} \tau$ binds x in τ (but not in σ). That is, $\mathit{fv}(\{x{:}B \mid \theta\}) = \mathit{fv}(\theta) \setminus \{x\}$, and $\mathit{fv}(x{:}\sigma \xrightarrow{\Phi} \tau) = \mathit{fv}(\sigma) \cup (\mathit{fv}(\tau) \setminus \{x\})$. Types are equivalent up to renaming of bound variables. We say that a type is *closed* if it contains no free variables.

4.1 Type Environment

A *type environment* Γ is a sequence of variable binding $x : \tau$, function name binding $\mathtt{F} : \tau$, and a FOL formula θ. The formulas θ are used to express the branch conditions (cf. [25, 28]). We often treat Γ as a set and write $x : \tau \in \Gamma$, $\theta \in \Gamma$, etc. to access its elements. We write $\mathit{dom}(\Gamma)$ for the variables and function names that are bound in Γ, that is, $\mathit{dom}(\Gamma) = \{\kappa \mid \kappa : \tau \in \Gamma\}$. (Here, κ ranges over variables and function names.) We assume that the (base-type) variables appearing free in the types and the formulas in Γ are in $\mathit{dom}(\Gamma)$, that is, $(\bigcup_{\theta\in\Gamma} \mathit{fv}(\theta) \cup \bigcup_{\kappa:\tau\in\Gamma} \mathit{fv}(\tau)) \subseteq \mathit{dom}(\Gamma)$.

$\llbracket\Gamma\rrbracket_{base}$ is the FOL formula denoting the assumptions about the base-type variables in Γ, and is defined as follows.

$$
\begin{array}{rcl}
\llbracket\varepsilon\rrbracket_{base} & = & \top \\
\llbracket\Gamma, x{:}\{u{:}B \mid \theta\}\rrbracket_{base} & = & \llbracket\Gamma\rrbracket_{base} \wedge \theta[x/u] \\
\llbracket\Gamma, \theta\rrbracket_{base} & = & \llbracket\Gamma\rrbracket_{base} \wedge \theta \\
\llbracket\Gamma, x{:}\tau\rrbracket_{base} & = & \llbracket\Gamma\rrbracket_{base} \quad \tau \text{ not refinement base type}
\end{array}
$$

We use the meta variable Δ for type environment containing only the function names, and Θ for the type environment containing only the variables (both base-type and function-type variables) and the formulas. Note that any type environment Γ can be split into the function name part and the variable part so that $\Gamma = \Delta, \Theta$.

4.2 Variable-to-value Substitution

We use the meta variable ρ to denote substitutions from variables to values (ρ does not map function names). (Recall that values are closed.) We define $\rho(e)$, $\rho(\theta)$, and $\rho(\tau)$ in the obvious way. We define $\rho(\Gamma)$ to be the environment with each $\theta \in \Gamma$ replaced by $\rho(\theta)$ and each $\kappa : \tau \in \Gamma$ replaced by $\kappa : \rho(\tau)$. We let $\rho\restriction_{base}$ denote the restriction of ρ to base type variables, and $\rho\restriction_{fn}$ denote the restriction of ρ to function type variables. Note that, because the formulas and the types only have base-type variables, substitution over the formulas can be restricted to base-type variables (*i.e.*, $\rho(\theta) = \rho\restriction_{base}(\theta)$ and $\rho(\tau) = \rho\restriction_{base}(\tau)$).

4.3 Semantics of Type and Effect

We define the semantics of type and effect for the purpose of defining the notion of valid oracles that can be used in the type system and for formalizing the soundness of the type system. Informally, the type and effect semantics $\llbracket\Theta \vdash \tau \mathbin{\&} \Phi\rrbracket_P$ is the set of expressions of the program P whose evaluation causes (finite and infinite) sequences of events in Φ, and if terminates, results in a value of the type τ, when the free variables in the expression are substituted the values of the types Θ. Figure 6 formally defines $\llbracket\Theta \vdash \tau \mathbin{\&} \Phi\rrbracket_P$. Here, $\rho \models_P \Theta$ denotes that 1.) $\mathit{dom}(\rho) = \mathit{dom}(\Theta)$, 2.) $\models \rho\restriction_{base}(\llbracket\Theta\rrbracket_{base})$, and 3.) $\forall x \in \mathit{dom}(\rho\restriction_{fn}).\rho(x) \in \llbracket\rho(\Theta(x))\rrbracket_P$.

Note that the definition uses the auxiliary notion $\llbracket\tau\rrbracket$ which represents the set of values of the type. As with the evaluation relations, we omit the subscript P and write $\llbracket\Theta \vdash \tau \mathbin{\&} \Phi\rrbracket$, $\llbracket\tau\rrbracket$, and $\rho \models \Theta$ when the program is clear from the context.

4.4 Oracle

We formalize oracle as follows.

Definition 4.1 (Oracle)**.** An *oracle* $\blacktriangleright_P$ is a 4-ary relation that satisfies $\Theta \blacktriangleright_P e : \tau \mathbin{\&} \Phi \Rightarrow e \in \llbracket\Theta \vdash \tau \mathbin{\&} \Phi\rrbracket_P$, for any expression e of P.

That is, an oracle is simply defined to be an entity that derives semantically correct judgements for the program's expressions. In fact, by the type soundness theorem (Theorem 4.1), our type system itself is a valid oracle. As before, we write $\Theta \blacktriangleright e : \tau \mathbin{\&} \Phi$ for $\Theta \blacktriangleright_P e : \tau \mathbin{\&} \Phi$ when P is clear from the context.

4.5 Typing Rules

Figure 7 shows the typing rules. A typing judgement for expressions is of the form $\Gamma \vdash e : \tau \mathbin{\&} \Phi$ expressing that the under the environment Γ, e has type τ with the effect Φ. We write $\Gamma \vdash e : \tau$ when there exists some Φ such that $\Gamma \vdash e : \tau \mathbin{\&} \Phi$. The type judgements are implicitly parameterized by the program being typed (used in the rule **Program**, and in **Oracle** for $\Theta \blacktriangleright e : \tau \mathbin{\&} \Phi$).

We implicitly assert that a type assigned to an expression conforms to the expression's simple type. More formally, the *simple-type shape* of τ is the simple type $\mathit{stsh}(\tau)$ defined such that $\mathit{stsh}(x{:}\sigma \xrightarrow{\Phi} \tau) = \mathit{stsh}(\sigma) \rightarrow \mathit{stsh}(\tau)$, and $\mathit{stsh}(\{u{:}B \mid \theta\}) = B$.

$$
\begin{array}{rcl}
e \in [\![\Theta \vdash \tau \,\&\, (\Xi, \Pi)]\!]_P & \Leftrightarrow & \forall \rho.\rho \vDash_P \Theta \Rightarrow \\
 & & \quad (\forall v, \varpi.\rho(e) \Downarrow_P v \,\&\, \varpi \Rightarrow v \in [\![\rho(\tau)]\!]_P \wedge \varpi \in \Xi) \\
 & & \quad \wedge (\forall \pi.\rho(e) \Uparrow_P \bot \,\&\, \pi \Rightarrow \pi \in \Pi) \\
c \in [\![\{u{:}B \mid \theta\}]\!]_P & \Leftrightarrow & c \in B \wedge \vDash \theta[c/u] \\
\lambda x.e \in [\![x{:}\tau \xrightarrow{\Phi} \sigma]\!]_P & \Leftrightarrow & e \in [\![x{:}\tau \vdash \sigma \,\&\, \Phi]\!]_P
\end{array}
$$

Figure 6: Semantics of type and effect.

$$
\frac{\Theta \blacktriangleright e : \tau \,\&\, \Phi}{\Delta, \Theta \vdash e : \tau \,\&\, \Phi}\ \textbf{Oracle}
\qquad
\frac{\Gamma \vdash e : \tau \,\&\, \Phi_1 \quad \Gamma \vdash e : \tau \,\&\, \Phi_2}{\Gamma \vdash e : \tau \,\&\, \Phi_1 \cap \Phi_2}\ \textbf{Comb}
\qquad
\frac{\Gamma \vdash e : \tau_1 \,\&\, \Phi_1 \quad \Gamma \vdash \tau_1 \leq \tau_2 \quad \Phi_1 \subseteq \Phi_2}{\Gamma \vdash e : \tau_2 \,\&\, \Phi_2}\ \textbf{Sub}
$$

$$
\frac{c \in B}{\Gamma \vdash c : \{u{:}B \mid u = c\} \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{Const}
\qquad
\frac{sty(x) = B}{\Gamma \vdash x : \{u{:}B \mid u = x\} \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{VaB}
\qquad
\frac{sty(x) \in \rightarrow}{\Gamma \vdash x : \Gamma(x) \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{VaF}
$$

$$
\frac{}{\Gamma \vdash \mathtt{F} : er(\Gamma(\mathtt{F})) \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{Fun}
\qquad
\frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash y : \tau_2 \quad ty(op) = x_1{:}\tau_1 \xrightarrow{\Phi_1} x_2{:}\tau_2 \xrightarrow{\Phi_2} \tau_3}{\Gamma \vdash x \; op \; y : \tau_3[x/x_1][y/x_2] \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{Op}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \Phi}{\Gamma \vdash \lambda x.e : x{:}\tau_1 \xrightarrow{\Phi} \tau_2 \,\&\, (\{\varepsilon\}, \varnothing)}\ \textbf{Lam}
\qquad
\frac{\Gamma \vdash x : z{:}\tau_1 \xrightarrow{\Phi} \tau_2 \quad \Gamma \vdash y : \tau_1}{\Gamma \vdash x\,y : \tau_2[y/z] \,\&\, (\{\underline{\mathbf{step}}\}, \varnothing) \cdot \Phi}\ \textbf{App}
$$

$$
\frac{\Gamma, x = \mathsf{true} \vdash e_1 : \tau \,\&\, \Phi_1 \quad \Gamma, x = \mathsf{false} \vdash e_2 : \tau \,\&\, \Phi_2}{\Gamma \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, \Phi_1 \cup \Phi_2}\ \textbf{If}
\qquad
\frac{\Gamma \vdash e : \tau \,\&\, \Phi \quad \vDash [\![\Gamma]\!]_{base} \Rightarrow \bot}{\Gamma \vdash e : \tau \,\&\, (\varnothing, \varnothing)}\ \textbf{Unreach}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \,\&\, \Phi_1 \\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \,\&\, \Phi_2\end{array}}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2 \,\&\, \Phi_1 \cdot \Phi_2}\ \textbf{Let}
\qquad
\frac{}{\Gamma \vdash \mathtt{ev}[\underline{\mathbf{a}}] : \mathtt{unit} \,\&\, (\{a\}, \varnothing)}\ \textbf{Event}
\qquad
\frac{\forall \mathtt{F}\ \tilde{x} = e \in P.\Gamma \vdash \lambda \tilde{x}.e : \Gamma(\mathtt{F})}{\Gamma \vdash \star}\ \textbf{Program}
$$

Figure 7: Typing rules.

Then, we assert that any type τ assigned to e in a type derivation satisfies $sty(e) = stsh(\tau)$.

We describe each rule. **Oracle** allows information derived by an external oracle to be used in the typing derivation. For example, consider the function below.

$$\mathsf{f}_0\ x = \mathtt{ev}[\underline{\mathbf{a}}]; \mathtt{if}\ x = 0\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{f}_0\ (x - 1)$$

A termination verifier for functional programs [20] may be used as an oracle to derive $\blacktriangleright\ \mathsf{f}_0 : \tau_{\mathsf{f}0} \,\&\, (\varnothing, \varnothing)$ where

$$\tau_{\mathsf{f}0} = x{:}\{u{:}\mathtt{int} \mid u \geq 0\} \xrightarrow{(\Sigma^*, \varnothing)} \mathtt{unit}$$

With this, **Oracle** can derive that $\mathsf{f}_0 : \tau_{\mathsf{f}0} \vdash \mathsf{f}_0 : \tau_{\mathsf{f}0} \,\&\, (\varnothing, \varnothing)$. Note that $\tau_{\mathsf{f}0}$ states that, given a non-negative integer argument, f_0 has no non-terminating event traces (*i.e.*, it terminates), but may have arbitrary terminating event traces (*i.e.*, the termination oracle says nothing beyond the fact that the function terminates on non-negative inputs).

Comb combines information from multiple derivations. While innocuous, the rule is essential for incorporating the facts derived by an oracle or combining the facts from multiple oracles. For example, as we show below, **App** and **Sub** can be used to derive that

$$\mathsf{f}_0{:}\tau_{\mathsf{f}0}, x{:}\{u{:}\mathtt{int} \mid u \geq 0\} \vdash \mathsf{f}_0\ x : \mathtt{unit} \,\&\, (\Sigma^*, \varnothing)$$

in the example above. That is, the call to f_0 with a non-negative integer argument terminates. Then, a safety oracle may be used to derive via **Oracle** (alternatively, the type system can derive such a safety judgement by itself):

$$\mathsf{f}_0{:}\tau_{\mathsf{f}0}, x{:}\{u{:}\mathtt{int} \mid u \geq 0\} \vdash \mathsf{f}_0\ x : \mathtt{unit} \,\&\, (\{\underline{\mathbf{step}}, \underline{\mathbf{a}}\}^*, \Sigma^\omega)$$

$$
\begin{array}{l}
er(\{u{:}B \mid \theta\}) = \{u{:}B \mid \theta\} \\
er(x{:}\sigma \xrightarrow{(\Xi, \Pi)} \tau) = x{:}\sigma \xrightarrow{(\Xi, \Sigma^\omega)} er(\tau)
\end{array}
$$

Figure 9: $er(\tau)$.

which says that the function call only generates either $\underline{\mathbf{step}}$ or $\underline{\mathbf{a}}$ event, if it terminates. Note that this judgement gives no information about the non-termination behavior. Then, **Comb** may combine the two judgements to derive:

$$\mathsf{f}_0{:}\tau_{\mathsf{f}0}, x{:}\{u{:}\mathtt{int} \mid u \geq 0\} \vdash \mathsf{f}_0\ x : \mathtt{unit} \,\&\, (\{\underline{\mathbf{step}}, \underline{\mathbf{a}}\}^*, \varnothing)$$

which says that the function call only generates terminating event traces consisting of $\underline{\mathbf{step}}$ and $\underline{\mathbf{a}}$.

Sub is the subsumption rule for subtypes and sub-effects. The subtyping rules are defined in Figure 8. The rule is an extension of the one used in the previous work on refinement type systems [6, 16, 19, 25, 28, 30] with the usual rule for sub-effecting that allow a larger effect to be used in place of a smaller effect.

Const, **VaB**, and **VaF** for typing constants and variables are straightforward extension of those from the previous work on refinement type systems. Here, $\rightarrow$ denotes the set of function-type simple types. Looking up a variable generates no non-terminating effects (*i.e.*, $\varnothing$) because the operation is guaranteed to terminate, whereas it generates an empty terminating event sequence (*i.e.*, $\{\varepsilon\}$).

Fun looks up the type of a recursive function name in the type environment. The rule is similar to **VaF**, except that the type in the conclusion is $er(\Gamma(\mathtt{F}))$ instead of $\Gamma(\mathtt{F})$. Informally, $er(\tau)$

"erases" the non-terminating parts of the top-level effects of τ by replacing them with Σ^ω. This prevents the type system from deriving non-trivial facts about non-terminating runs by itself (*i.e.*, without the help of an oracle), and is needed to ensure the type system's soundness. For example, suppose $er(\Gamma(\mathtt{F}))$ in the conclusion of **Fun** was replaced by $\Gamma(\mathtt{F})$. Then, for the program $\{\dots, \mathsf{loop}\ x = \mathsf{loop}\ x\}$, we would be able to derive $\Gamma \vdash \star$ where $\Gamma(\mathsf{loop}) = x{:}\mathtt{int} \xrightarrow{(\Sigma^*,\varnothing)} \mathtt{unit}$, which says that a call to loop generates no infinite event traces, that is, it terminates. This is obviously incorrect as any call to loop diverges. The erasure prevents such unsound non-termination effects from occurring. We formally define $er(\tau)$ in Figure 9.

Op types constant operator applications. Here, $ty(op)$ is the *sound constant operator type* of op. Formally, a sound constant operator type of *op* is defined to be a closed type of the form

$$x_1{:}B_1 \xrightarrow{(\{\varepsilon\},\varnothing)} x_1{:}B_2 \xrightarrow{(\{\varepsilon\},\varnothing)} \{u{:}B_3 \mid \theta\}$$

that satisfy the following:

- *op* is a binary operator from the arguments of the type B_1 and B_2 to the return value of the type B_3.
- For constants c_1 and c_2 of the type B_1 and B_2 and $c_3 = [\![op]\!](c_1, c_2), \vDash \theta[c_1/x_1][c_2/x_2][c_3/u]$.

For example, $x : \mathtt{int} \xrightarrow{(\{\varepsilon\},\varnothing)} y{:}\mathtt{int} \xrightarrow{(\{\varepsilon\},\varnothing)} \{u{:}\mathtt{int} \mid u = x + y\}$ is a sound type for the integer addition operator +

Lam types function definitions, and **App** types function applications. The rules are extensions of the ones in the previous work on refinement type systems with the usual type-and-effect approach of recording and discharging the latent effect. As remarked in Section 3, the $\underline{\mathbf{step}}$ event guard is used to prevent non-productive non-terminating runs.

If types conditional branches. Per previous work on refinement type systems, each branch is typed with the assumption about the branch condition added to the type environment (*i.e.*, $x = \mathtt{true}$ and $x = \mathsf{false}$). **Unreach** allows the type system to derive that e generates no effects in an unreachable context (*i.e.*, when $[\![\Gamma]\!]_{base}$ is unsatisfiable).

Let types let bindings. Note that the effect Φ_1 of e_1 is appended to the effect Φ_2 of e_2. **Event** types event raising operations and is self-explanatory. Continuing the function f_0 example from above, we may use **Let** and **Event** to derive that

$$\mathsf{f}_0{:}\tau_{\mathsf{f}0}, x{:}\{u{:}\mathtt{int} \mid u \geq 0\} \vdash \mathsf{f}_0\, x; \mathtt{ev}[\underline{\mathbf{b}}] : \mathtt{unit}\ \&\ (\Xi_0, \varnothing)$$

where $\Xi_0 = \{\underline{\mathbf{step}}, \underline{\mathbf{a}}\}^* \cdot \{\underline{\mathbf{b}}\}$. That is, $\mathsf{f}_0\, x; \mathtt{ev}[\underline{\mathbf{b}}]$ only generates finite sequences of $\underline{\mathbf{step}}$ and $\underline{\mathbf{a}}$ that end in $\underline{\mathbf{b}}$, which is the property $(\underline{\mathbf{step}} \vee \underline{\mathbf{a}})\, \mathsf{U}\, \underline{\mathbf{b}}$ in LTL.

Finally, **Program** types the program P (implicit in the rules) by asserting that each of the recursive functions have types of their fixpoints. We state the soundness of the type system.

Theorem 4.1 (Soundness)**.** Suppose $\Delta \vdash \star$, $dom(\Theta) = fv(e)$, and $\Delta, \Theta \vdash e : \tau\ \&\ \Phi$. Then, $e \in [\![\Theta \vdash \tau\ \&\ \Phi]\!]$

Proof. Please see Appendix A. □

The following is immediate from **Oracle**, and states that the type system is complete relative to the oracle.

Theorem 4.2 (Relative Completeness)**.** Suppose $\Theta \blacktriangleright e : \tau\ \&\ \Phi$. Then, $\Delta, \Theta \vdash e : \tau\ \&\ \Phi$.

We note that, without **Oracle**, the type system cannot prove non-trivial liveness properties (*i.e.*, properties about non-terminating runs) in presence of recursive functions. For example, let $P = \{\mathtt{H}\, y = \mathtt{if}\ y \leq 0\ \mathtt{then}\ ()\ \mathtt{else}\ \mathtt{H}\,(y-1)\}$ and suppose that we would like to prove that the call to H with any integer argument terminates. The property can be expressed as the type $\tau_{\mathtt{H}} = x{:}\mathtt{int} \xrightarrow{(\Sigma^*,\varnothing)} \mathtt{unit}$, and we try to derive that $\mathtt{H} : \tau_{\mathtt{H}} \vdash \star$. But, this cannot be done without **Oracle** because any occurrence of H in the derivation will have its type's top-most non-terminating effect "erased". (Trivially, soundness holds even if the **Oracle** rule is removed from the type system.)

5. Examples

In this section we illustrate how our technique operates through a variety of instantiations on examples. Note there are no existing techniques to prove that the following properties hold over their respective programs. For illustrative purposes, some examples are first order however, of course, the power of our technique is that it applies to higher order programs. In this section we have made all events explicit.

In the REDUCE example in Figure 10, we rely on a termination oracle to tell us that explore terminates when it's given a certain type of R:

$$\begin{gathered}\Gamma, \mathsf{R} : \{i : \mathtt{int} \mid \mathsf{true}\} \xrightarrow{(\underline{\mathbf{step}}^+,\bot)} \{j : \mathtt{int} \mid j < i\} \\ \vdash \mathsf{explore} \times \mathsf{R} : \mathtt{unit}\ \&\ (\top, \bot)\end{gathered}$$

We also obtain safety information from the type system:

$$\begin{gathered}\Gamma, \mathsf{R} : \{i : \mathtt{int} \mid \mathsf{true}\} \xrightarrow{(\underline{\mathbf{step}}^+,\bot)} \{j : \mathtt{int} \mid j < i\} \\ \vdash \mathsf{explore} \times \mathsf{R} : \mathtt{unit}\ \&\ ((\underline{\mathbf{explore}} \mid \underline{\mathbf{step}})^* \cdot \underline{\mathbf{done}}, \top)\end{gathered}$$

Via the **Sub** rule, we can rewrite the combination (**Comb**) of these two rules and conclude that the body of main has effect $\mathsf{F}(\underline{\mathbf{done}})$. Note that the termination oracle here need only be used to show that explore is not called infinitely often (whole program termination reasoning is not needed).

The RUMBLE example illustrates modular reasoning. Here we again use a termination oracle to tell us that rumble terminates regardless of input, but we use it to refine the safety information differently. Our type system can conclude that the innermost application in main has (finitely many) $\underline{\mathbf{step}}$ events:

$$\Gamma \vdash \mathsf{rumble\ b\ a} : \dots\ \&\ ((\underline{\mathbf{step}} \mid \underline{\mathbf{rumble}})^+, \top)$$

We use the **Comb** rule to combine this with information from the termination oracle to obtain

$$\Gamma \vdash \mathsf{rumble\ b\ a} : \dots\ \&\ ((\underline{\mathbf{step}} \mid \underline{\mathbf{rumble}})^+, \bot).$$

(We have omitted the refinement typing, which is not relevant here.) We then use the **App**, **Oracle** and **Comb** rules again to obtain

$$\Gamma \vdash \mathsf{rumble\ a\ (rumble\ b\ a)} : \dots\ \&\ ((\underline{\mathbf{step}} \mid \underline{\mathbf{rumble}})^+, \bot).$$

Finally, we consider the application of print to obtain

$$\Gamma \vdash \mathsf{main}\dots : \dots\ \&\ ((\underline{\mathbf{step}} \mid \underline{\mathbf{rumble}})^+ \cdot \underline{\mathbf{print}}, \bot)$$

which entails $\mathsf{F}(\underline{\mathbf{print}})$. A benefit of our type system is illustrated here: information from an oracle can be reused.

EVENTUALLY GLOBALLY demonstrates nesting of G within F. To prove this example, we combine a few facts:

1. A termination oracle tells us that bar terminates.
2. We can indicate that bar always returns a non-positive number with the dependent typing:
   $\Gamma \vdash \mathsf{bar} : \mathtt{int} \xrightarrow{((\underline{\mathbf{bar}} | \underline{\mathbf{step}})^+,\top)} \{j : \mathtt{int} \mid j \leq 0\}\ \&\ \dots$
3. The type-and-effect system can conclude, with the help of a *non*termination oracle that foo x does not terminate when $\mathsf{x} \leq 0$ and that its infinite behavior will be $(\underline{\mathbf{foo}} \mid \underline{\mathbf{step}})^\omega$ which is a fixpoint of $\alpha = \underline{\mathbf{foo}} \wedge \mathsf{X}\alpha$:
   $\Gamma, \mathsf{x} : \{j : \mathtt{int} \mid j \leq 0\} \quad \vdash \quad \mathsf{foo\ x} : \dots\ \&\ (\bot, (\underline{\mathbf{foo}} \mid \underline{\mathbf{step}})^\omega)$

REDUCE	RUMBLE	EVENTUALLY GLOBAL	ALTERNATE INEVITABILITY

let rec done _ =
ev[**done**]
and reduce x R =
if x ≤ 0 **then** x
else R x
 	
let rec explore x R =
ev[**explore**] ;
let y = reduce x R **in**
if y ≤ 0 **then** done ()
else explore y R
 	
let main() =
let t = * **in**
explore t (λx. x - 2)
	let rec print_int x =
	ev[**print**] ; ...
	and rumble x y =
	ev[**rumble**] ;
	if x < y **then**
	if * **then**
	rumble (x+1) y
	else rumble x (y-1)
	else x
	
	let main() =
	let a = * **in**
	let b = * **in**
	print (rumble a
	(rumble b a))
		let rec halt _ =
		ev[**halt**]
		and bar x = ev[**bar**] ;
		if x > 0 **then** bar (x-2)
		else x
		
		and foo x = ev[**foo**] ;
		if (x ≤ 0) then foo x
		else halt ()
		
		let main () =
		let t = * **in**
		if * **then** foo 0
		else foo (bar t)
			let app f x i = f x (λ t. t - i)
			let ha1 _ = ev[**ha1**]
			let ha2 _ = ev[**ha2**]
			let rec walk x f = ev[**walk**] ;
			if x = 0 **then** x **else** walk (f x) f
			and run x f = ev[**run**] ;
			if x = 0 **then** x
			else run (f (f x)) f
			and life x =
			if * **then**
			ev[**p**];
			if x<0 **then** ha1 (app walk x 1)
			else ha2 (app run x 1)
			else life x
			let main() = life *

$\Phi = \mathsf{F}(\textbf{done})$	$\Phi = \mathsf{F}(\textbf{print})$	$\Phi = \mathsf{FG}(\textbf{foo} \vee \textbf{step})$	$\Xi = \mathsf{G}(\textbf{p} \Rightarrow \mathsf{X}(\textbf{walk}\,\mathsf{U}\,\textbf{ha1} \vee \textbf{run}\,\mathsf{U}\,\textbf{ha2}))$

Figure 10: Example programs and corresponding properties (abbreviated in LTL).

We combine all of this information together to show that the foo call sites in main satisfy $\mathsf{F}(\mathsf{G}(\textbf{foo} \vee \textbf{step}))$.

For ALTERNATE INEVITABILITY, ultimately we need to show that the finite traces of life x satisfies $\mathsf{G}(\textbf{p} \Rightarrow \mathsf{X}(\textbf{walk}\,\mathsf{U}\,\textbf{ha1} \vee \textbf{run}\,\mathsf{U}\,\textbf{ha2}))$. We elide detail related to **step** events. We begin with the inner if/else in life. In the then branch, app applies the function walk on argument x and on a function that subtracts by one. An oracle can show that this terminates:

$$\Gamma \vdash \mathsf{walk} : \\ \mathsf{x} : \mathtt{int} \to \mathsf{f} : (i : \mathtt{int} \to \{j : \mathtt{int} \mid j < i\}) \xrightarrow{(\top,\bot)} \mathtt{int}$$

Similar for run. Combining this information with a safety judgment that walk generates $\textbf{walk}^*$, and using the **App** rule to append the **ha1** event, we have summarized the effects of the then branch. We do the same for the else branch, and then with the semi-colon rule, we obtain $(\textbf{p} \Rightarrow \mathsf{X}(\textbf{walk}\,\mathsf{U}\,\textbf{ha1} \vee \textbf{run}\,\mathsf{U}\,\textbf{ha2}))$ for the then branch of the outer if/else expression. The else branch is another call to life x, so we look for a fixpoint to

$$\alpha = \alpha \vee (\textbf{p} \Rightarrow \mathsf{X}(\textbf{walk}\,\mathsf{U}\,\textbf{ha1} \vee \textbf{run}\,\mathsf{U}\,\textbf{ha2}))$$

One solution is $\mathsf{G}(\textbf{p} \Rightarrow \mathsf{X}(\textbf{walk}\,\mathsf{U}\,\textbf{ha1} \vee \textbf{run}\,\mathsf{U}\,\textbf{ha2}))$. Something critical here was that we had complete knowledge of all instances where **p** was generated. Because our type-and-effect system can maintain precise descriptions of the event traces, we can build this up syntactically over the body of life.

6. Related work

To the best of our knowledge, our work is the first technique that is able to prove temporal properties of higher-order, infinite-data programs. In Sections 1 and 2 we summarized previous efforts that are restricted in one way or another (*e.g.* only finite-data, safety-only, termination-only, or works that are restricted to first-order programs). We now discuss some other related works.

In our work we strive to achieve *exogenous* verification in that we are attempting to verify an overall external behavior of a program, as opposed to *endogenous* verification (*e.g.* Floyd/Hoare Logic), which is more directly connected to system internals such as program location. However, we use an endogenous type-and-effect system, combined with generalizing composed effects. In their work, Barringer *et al.* [3] extend temporal logic with a composition operator. Their work, however, is geared toward compositional user-provided (endogenous) specifications rather than our goal of proving an overall (exogenous) specification, by composing together pieces of reasoning.

We are not the first to suggest a type-and-effect system for verifying temporal properties. Skalka *et al.* [26, 27] describe a type-and-effect system where event traces are effects, similar to our system. However, unlike our system, they use the type-and-effect system only to infer an over-approximation of the program's actual event traces, which is then checked against the target temporal property by an external model checker.[2] Their approach precludes the possibility of verifying non-trivial liveness properties, because a type-and-effect system alone is inherently "safety", and having an up-front type-and-effect abstraction can result in losing the information needed for liveness reasoning. By contrast, our type-and-effect system allows the use of temporal property verifiers inside the type derivation as oracles, enabling compositional verification of non-trivial temporal properties.

A different approach to using a type system for temporal property verification has been proposed by Kobayashi and Ong [18]. In their approach, the type system is used internally to define a parity game (*i.e.* a "move" in the game is defined as a typability relation), and the verification problem is reduced to the problem of finding a winning strategy to the game. Our approach is conceptually the dual of theirs, and allows verifiers to be used as oracles inside the type system. Also, unlike our approach, their approach is limited to the verification of finite data programs (given in CPS), and it may be difficult to extend the approach to infinite data programs.

7. Conclusion

We have introduced the first technique that enables us to verify temporal properties of higher-order, infinite-data programs. Our type-and-effect system accomplishes this by decomposing the program, not only into expressions, but also based on the behavior of each expression's finite versus infinite event behaviors. The type system itself is strong enough to derive temporal safety properties which can be combined with liveness information from oracles. We believe this work will serve as a theoretical foundation toward automatic verification of not only higher-order programs, but also provides a new route to more compositional verification of first-order programs.

[2] For this reason, they use BPA as the concrete model of effects, whereas we keep the effect representation intentionally abstract.

References

[1] ALUR, R., AND CHAUDHURI, S. Temporal reasoning for procedural programs. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)* (2010), vol. 5944, pp. 45–60.

[2] BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software* (2001), vol. 2057, pp. 103–122.

[3] BARRINGER, H., KUIPER, R., AND PNUELI, A. Now you may compose temporal logic specifications. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC '84)* (1984), pp. 51–63.

[4] BEYENE, T., POPEEA, C., AND RYBALCHENKO, A. Solving existentially quantified horn clauses. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'11)* (2013).

[5] BEYER, D., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. The software model checker blast. *STTT 9*, 5-6 (2007), 505–525.

[6] BHARGAVAN, K., FOURNET, C., AND GORDON, A. D. Modular verification of security protocol code by typing. In *The 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)* (2010), pp. 445–456.

[7] CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)* (2000), vol. 1855, pp. 154–169.

[8] COOK, B., GOTSMAN, A., PODELSKI, A., RYBALCHENKO, A., AND VARDI, M. Y. Proving that programs eventually do something good. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)* (2007), pp. 265–276.

[9] COOK, B., AND KOSKINEN, E. Making prophecies with decision predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)* (2011), ACM, pp. 399–410.

[10] COOK, B., AND KOSKINEN, E. Reasoning about nondeterminism in programs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (2013), ACM.

[11] COOK, B., KOSKINEN, E., AND VARDI, M. Temporal verification as a program analysis task [extended version]. *Formal Methods in System Design* (2012).

[12] COOK, B., KOSKINEN, E., AND VARDI, M. Y. Temporal property verification as a program analysis task. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011), vol. 6806, pp. 333–348.

[13] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)* (2006), pp. 415–426.

[14] COUSOT, P., AND COUSOT, R. Inductive definitions, semantics and abstract interpretation. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1992), R. Sethi, Ed., ACM Press, pp. 83–94.

[15] GASTIN, P., AND ODDOUX, D. Fast LTL to Büchi automata translation. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'01)* (2001), pp. 53–65.

[16] JHALA, R., MAJUMDAR, R., AND RYBALCHENKO, A. HMC: Verifying functional programs using abstract interpreters. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011).

[17] JOHNSSON, T. Lambda lifting: Transforming programs to recursive equations. In *FPCA* (1985), pp. 190–203.

[18] KOBAYASHI, N., AND ONG, C.-H. L. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS'09)* (2009), pp. 179–188.

[19] KOBAYASHI, N., SATO, R., AND UNNO, H. Predicate abstraction and cegar for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)* (2011), pp. 222–233.

[20] KUWAHARA, T., TERAUCHI, T., UNNO, H., AND KOBAYASHI, N. Automatic termination verification for higher-order functional programs. In *Proceedings of the 22nd European Symposium on Programming (ESOP'14)* (2014), vol. 7792, pp. 1–20.

[21] LEDESMA-GARZA, R., AND RYBALCHENKO, A. Binary reachability analysis of higher order functional programs. In *Proceedings of the 19th International Symposium on Static Analysis (SAS'12)* (2012), vol. 7460, pp. 388–404.

[22] LEROY, X., AND GRALL, H. Coinductive big-step operational semantics. *Inf. Comput. 207*, 2 (2009), 284–304.

[23] MCMILLAN, K. L. Lazy abstraction with interpolants. In *CAV'06* (2006), T. Ball and R. B. Jones, Eds., vol. 4144, pp. 123–136.

[24] PODELSKI, A., AND RYBALCHENKO, A. A complete method for the synthesis of linear ranking functions. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)* (2004), vol. 2937, Springer, pp. 239–251.

[25] RONDON, P. M., KAWAGUCHI, M., AND JHALA, R. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)* (2008), ACM, pp. 159–169.

[26] SKALKA, C., AND SMITH, S. F. History effects and verification. In *Proceedings of Programming Languages and Systems: Second Asian Symposium, (APLAS 2004)* (2004), W.-N. Chin, Ed., vol. 3302, Springer, pp. 107–128.

[27] SKALKA, C., SMITH, S. F., AND HORN, D. V. Types and trace effects of higher order programs. *J. Funct. Program. 18*, 2 (2008), 179–249.

[28] TERAUCHI, T. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)* (2010), ACM, pp. 119–130.

[29] THOMAS, W. Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA, 1990, ch. Automata on Infinite Objects, pp. 133–191.

[30] UNNO, H., TERAUCHI, T., AND KOBAYASHI, N. Automating relatively complete verification of higher-order functional programs. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)* (2013), pp. 75–86.

[31] XI, H., AND PFENNING, F. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)* (1999), pp. 214–227.

A. Proof of Theorem 4.1

We add to the type system an "the-all-knowing oracle" rule that can derive arbitrary semantically true judgements:

$$\frac{e \in [\![\Theta \vdash \tau \,\&\, \Phi]\!]}{\Delta, \Theta \vdash e : \tau \,\&\, \Phi}\ \textbf{All}$$

We state a few lemmas regarding subtyping.

Lemma A.1 (≤ Transitivity)**.** Suppose $\Gamma \vdash \tau_1 \le \tau_2$ and $\Gamma \vdash \tau_2 \le \tau_3$. Then, $\Gamma \vdash \tau_1 \le \tau_3$.

Lemma A.2 (≤ Substitution)**.** Suppose $\Delta, \Theta \vdash \tau \le \sigma$, and $\rho \vDash \Theta$. Then, $\Delta \vdash \rho(\tau) \le \rho(\sigma)$.

Lemma A.3. Suppose $\Gamma, x{:}\tau \vdash \sigma' \le \sigma$ and $\Gamma \vdash \tau' \le \tau$. Then, $\Gamma, x{:}\tau' \vdash \sigma' \le \sigma$.

Lemma A.4. Suppose $\Gamma, x{:}\tau \vdash e : \tau \,\&\, \Phi$ and $\Gamma \vdash \tau' \le \tau$. Then, $\Gamma, x{:}\tau' \vdash e : \tau \,\&\, \Phi$.

Lemma A.5 (≤ Semantic Correspondence I)**.** Suppose $\Delta, \Theta \vdash \tau \le \tau'$ and $\rho \vDash \Theta$. Then, $[\![\rho(\tau)]\!] \subseteq [\![\rho(\tau')]\!]$.

Proof. We prove by induction on the structure of τ.

Case $\tau = \{u{:}B \mid \theta\}$
By Lemma A.2, $\Delta \vdash \rho(\tau) \le \rho(\tau')$. By Lemma A.1 and inspection of the subtyping rules, it must be the case that $\tau' = \{u{:}B \mid \theta'\}$ and $\vDash \rho(\theta) \Rightarrow \rho(\theta')$. Therefore, $[\![\rho(\tau)]\!] \subseteq [\![\rho(\tau')]\!]$.

Case $\tau = x{:}\sigma_1 \xrightarrow{\Phi} \sigma_2$
By Lemma A.2, $\Delta \vdash \rho(\tau) \le \rho(\tau')$. By Lemma A.1 and inspection of the subtyping rules, it must be the case that $\tau' = x{:}\sigma_1' \xrightarrow{\Phi'} \sigma_2'$, $\Delta \vdash \rho(\sigma_1') \le \rho(\sigma_1)$, $\Delta, x{:}\rho(\sigma_1') \vdash \rho(\sigma_2) \le \rho(\sigma_2')$, and $\Phi \subseteq \Phi'$.
Let $\lambda x.e \in [\![\rho(\tau)]\!]$. By definition, $e \in [\![x{:}\rho(\sigma_1) \vdash \rho(\sigma_2) \,\&\, \Phi]\!]$. It suffices to show that $e \in [\![x{:}\rho(\sigma_1') \vdash \rho(\sigma_2') \,\&\, \Phi']\!]$ (because then $\lambda x.e \in [\![\rho(\tau')]\!]$). Suppose $v \in [\![\rho(\sigma_1')]\!]$. Then, by induction hypothesis, $v \in [\![\rho(\sigma_1)]\!]$. Therefore, if $e[v/x] \Downarrow v' \,\&\, \varpi$ then $v' \in [\![\rho(\sigma_2)[v/x]]\!]$. By induction hypothesis, the latter implies that $v' \in [\![\rho(\sigma_2')[v/x]]\!]$. Therefore, because $\Phi \subseteq \Phi'$, it follows that $e \in [\![x{:}\rho(\sigma_1') \vdash \rho(\sigma_2') \,\&\, \Phi']\!]$.

□

Lemma A.6 (≤ Semantic Correspondence II)**.** Suppose $\Delta, \Theta \vdash \tau \le \tau'$ and $\Phi \subseteq \Phi'$. Then, $[\![\Theta \vdash \tau \,\&\, \Phi]\!] \subseteq [\![\Theta \vdash \tau' \,\&\, \Phi']\!]$.

Proof. Suppose $\rho \vDash \Theta$, $\rho(e) \Downarrow v \,\&\, \varpi$, and $v \in [\![\rho(\tau)]\!]$. Then, by Lemma A.5, $v \in [\![\rho(\tau')]\!]$. Therefore, the statement follows. □

Lemma A.7 (Substitution)**.** Suppose $\Delta, \Theta \vdash e : \tau \,\&\, \Phi$ and $\rho \vDash \Theta$. Then, $\Delta \vdash \rho(e) : \rho(\tau) \,\&\, \Phi$.

Proof. By induction on the derivation of $\Delta, \Theta \vdash e : \tau \,\&\, \Phi$. □

Lemma A.8 (Preservation)**.** Suppose

- $\Delta \vdash \star$,
- $\Delta \vdash e : \tau \,\&\, (\Xi, \Pi)$,
- e is closed, and
- $e \Downarrow v \,\&\, \varpi$

Then, $\Delta \vdash v : \tau$ and $\varpi \in \Xi$

Proof. By induction on the derivation of $e \Downarrow v \,\&\, \varpi$. □

Lemma A.9 (Soundness Part I : Erasure)**.** Suppose

(1) $\Delta \vdash \star$,
(2) *dom*(Θ) = *fv*(e), and
(3) $\Delta, \Theta \vdash e : \tau \,\&\, (\Xi, \Pi)$

Then, $e \in [\![\Theta \vdash er(\tau) \,\&\, (\Xi, \Sigma^\omega)]\!]$.

Proof. First, note that the statement is equivalent to the following.

(a) Suppose (1), (2), (3), $\rho \vDash \Theta$, and $\rho(e) \Downarrow v \,\&\, \varpi$. Then, $\varpi \in \Xi$ and $v \in [\![\rho(er(\tau))]\!]$.

By Lemma A.7, $\Delta \vdash \rho(e) : \rho(\tau) \,\&\, (\Xi, \Pi)$, and so by Lemma A.8, $\Delta \vdash v : \rho(\tau)$ and $\varpi \in \Xi$. Therefore, (a) is implied by the following.

(b) Suppose (1), $\rho \vDash \Theta$, and $\Delta \vdash v : \rho(\tau)$. Then, $v \in [\![\rho(er(\tau))]\!]$.

We prove (b) by induction on the structure of τ.

Case $\tau = \{u{:}B \mid \theta\}$
We have that $\rho(er(\tau)) = \rho(\tau) = \{u{:}B \mid \rho(\theta)\}$. By the definition of values, $v = c$ for some constant c. Then, by **Const**, **Sub**, and Lemma A.1 and the definition of sound constant type, $\vDash u = c \Rightarrow \rho(\theta)$. Therefore, $c \in [\![\rho(\theta)]\!]$.

Case $\tau = x{:}\sigma' \xrightarrow{(\Xi',\Pi')} \tau'$
We have that $\rho(er(\tau)) = x{:}\rho(\sigma') \xrightarrow{(\Xi',\Sigma^\omega)} \rho(er(\tau'))$. It must be case that $v = \lambda x.e'$ for some e' and x. By inspection of the typing rules, either **All**, **Oracle**, or **Lam** must have been applied at v.

Case All or Oracle
By **Sub** and Lemma A.1, it must be the case that $v \in [\![\varepsilon \vdash \tau' \,\&\, \Phi]\!]$ such that $\Delta \vdash \tau' \le \rho(\tau)$.
We have that $v \in [\![\tau']\!]$. Therefore, by Lemma A.5, $v \in [\![\rho(\tau)]\!]$. It is easy to see that $\Delta \vdash \rho(\tau) \le \rho(er(\tau))$. Therefore, by Lemma A.5 again, $v \in [\![\rho(er(\tau))]\!]$.

Case Lam
By **Sub** and Lemma A.1, it must be the case that $\Delta, x : \sigma'' \vdash e' : \tau'' \,\&\, (\Xi'', \Pi'')$ where

- $\Delta \vdash \rho(\sigma') \le \sigma''$
- $\Delta, x{:}\rho(\sigma') \vdash \tau'' \le \rho(\tau')$
- $(\Xi'', \Pi'') \subseteq (\Xi', \Pi')$

By Lemma A.4 and **Sub**, we have $\Delta, x : \rho(\sigma') \vdash e' : \rho(er(\tau')) \,\&\, (\Xi', \Sigma^\omega)$. Therefore, by induction hypothesis (on $\rho(er(\tau'))$), we have that

$$e' \in [\![x{:}\rho(\sigma') \vdash \rho(er(\tau')) \,\&\, (\Xi', \Sigma^\omega)]\!]$$

Therefore,

$$\lambda x.e' \in [\![x{:}\rho(\sigma') \xrightarrow{(\Xi',\Sigma^\omega)} \rho(er(\tau'))]\!]$$

□

We are now ready to prove the soundness theorem.

Theorem 4.1: Suppose

- $\Delta \vdash \star$,
- *dom*(Θ) = *fv*(e), and
- $\Delta, \Theta \vdash e : \tau \,\&\, \Phi$

Then, $e \in [\![\Theta \vdash \tau \,\&\, \Phi]\!]$.

Proof. We prove by induction on the derivation of $\Delta, \Theta \vdash e : \tau \& \Phi$.

Case the last rule is Oracle
Immediate from Definition 4.1 and **Oracle**.

Case the last rule is Comb
It must be the case that $\Phi = \Phi_1 \cap \Phi_2$ where

$$\frac{\Delta, \Theta \vdash e : \tau \,\&\, \Phi_1 \quad \Delta, \Theta \vdash e : \tau \,\&\, \Phi_2}{\Delta, \Theta \vdash e : \tau \,\&\, \Phi_1 \cap \Phi_2}$$

By induction hypothesis, we have that

- $e \in [\![\Theta \vdash \tau \,\&\, \Phi_1]\!]$
- $e \in [\![\Theta \vdash \tau \,\&\, \Phi_2]\!]$

Therefore, $e \in [\![\Theta \vdash \tau \,\&\, \Phi_1 \cap \Phi_2]\!]$.

Case the last rule is Sub

We have

$$\frac{\Delta, \Theta \vdash e : \tau' \,\&\, \Phi' \quad \Delta, \Theta \vdash \tau' \leq \tau \quad \Phi' \subseteq \Phi}{\Delta, \Theta \vdash e : \tau \,\&\, \Phi}$$

By induction hypothesis, we have that

$$e \in [\![\Theta \vdash \tau' \,\&\, \Phi']\!]$$

Therefore, by Lemma A.6, $e \in [\![\Theta \vdash \tau \,\&\, \Phi]\!]$.

Case the last rule is Const

We have $e = c$ and

$$\frac{c \in B}{\Delta, \Theta \vdash c : \{u{:}B \mid u = c\} \,\&\, (\{\varepsilon\}, \varnothing)}$$

Suppose $\rho \vDash \Theta$. Then, because $c \in B$,

$$\rho(c) = c \in [\![\{u{:}B \mid u = c\}]\!] = [\![\rho(\{u{:}B \mid u = c\})]\!]$$

Therefore, by **st-Val**,

$$c \in [\![\Theta \vdash \{u{:}B \mid u = c\} \,\&\, (\{\varepsilon\}, \varnothing)]\!]$$

Case the last rules is VaB

We have $e = x$ and

$$\frac{\mathit{sty}(x) = B}{\Delta, \Theta \vdash x : \{u{:}B \mid u = x\} \,\&\, (\{\varepsilon\}, \varnothing)}$$

Suppose $\rho \vDash \Theta$. Then, because $\mathit{sty}(x) \in B$, we have

$$\rho(x) \in [\![\rho(\{u{:}B \mid u = x\})]\!]$$

Therefore, by **st-Val**, we have

$$x \in [\![\Theta \vdash \{u{:}B \mid u = x\} \,\&\, (\{\varepsilon\}, \varnothing)]\!]$$

Case the last rule is VaF

We have $e = x$ and

$$\frac{\mathit{sty}(x) \in \rightarrow}{\Delta, \Theta \vdash x : \Theta(x) \,\&\, (\{\varepsilon\}, \varnothing)}$$

Suppose $\rho \vDash \Theta$. Then, because $\mathit{sty}(x) \in \rightarrow$,

$$\rho(x) \in [\![\rho(\Theta(x))]\!]$$

Therefore, by **st-Val**, $x \in [\![\Theta \vdash \Theta(x) \,\&\, (\{\varepsilon\}, \varnothing)]\!]$.

Case the last rule is Fun

Immediate by Lemma A.9 and **st-Fun**.

Case the last rule is Op

We have $e = x \; \mathit{op} \; y$ and

$$\frac{\begin{array}{c} \mathit{ty}(\mathit{op}) = x_1{:}\tau_1 \xrightarrow{\Phi_1} x_2{:}\tau_2 \xrightarrow{\Phi_2} \tau_3 \\ \Delta, \Theta \vdash x : \tau_1 \,\&\, \Phi_1' \quad \Delta, \Theta \vdash y : \tau_2 \,\&\, \Phi_2' \end{array}}{\Delta, \Theta \vdash x \; \mathit{op} \; y : \tau_3[x/x_1][y/x_2] \,\&\, (\{\varepsilon\}, \varnothing)}$$

By the definition of the sound constant operator type, $\tau_1 = \{u{:}B_1 \mid \top\}$ and $\tau_2 = \{u{:}B_2 \mid \top\}$, and $\tau_3 = \{u{:}B_3 \mid \theta\}$ for some B_1, B_2,B_3, and θ.

Suppose $\rho \vDash \Theta$. By simple typing of x and y (or by induction hypothesis), we have $\rho(x) \in B_1$ and $\rho(y) \in B_2$. By the definition of sound constant operator type, if $\rho(x) \; \mathit{op} \; \rho(y) \Downarrow c \,\&\, \varepsilon$, then $c \in B_3$ and $\vDash \theta[\rho(x)/x_1][\rho(y)/y_2][c/u]$. Because $\mathit{fv}(\theta) \subseteq \{u, x_1, x_2\}$, it follows that

$$c \in [\![\rho(\{u{:}B_3 \mid \theta\}[x/x_1][y/x_2])]\!]$$

Therefore, by **st-Op**, $x \; \mathit{op} \; y \in [\![\Theta \vdash \tau_3[x/x_1][y/x_2] \,\&\, (\{\varepsilon\}, \varnothing)]\!]$.

Case the last rule is Lam

We have $e = \lambda x.e'$ and $\tau = x{:}\sigma' \xrightarrow{\Phi'} \tau'$ such that

$$\frac{\Delta, \Theta, x : \sigma' \vdash e' : \tau' \,\&\, (\Xi, \Pi)}{\Delta, \Theta \vdash \lambda x.e' : x{:}\sigma' \xrightarrow{(\Xi,\Pi)} \tau' \,\&\, (\{\varepsilon\}, \varnothing)}$$

Suppose $\rho \vDash \Theta$ and $v \in [\![\rho(\sigma')]\!]$. Then, $\rho \cup \{x \mapsto v\} \vDash \Theta, x{:}\sigma'$. Therefore, the following holds from the induction hypothesis $e' \in [\![\Theta, x : \sigma' \vdash \tau' \,\&\, (\Xi, \Pi)]\!]$.

- If $\rho(e')[v/x] \Downarrow v' \,\&\, \varpi$ then $v' \in \rho(\tau')[v/x]$ and $\varpi \in \Xi$.
- If $\rho(e')[v/x] \Uparrow \bot \,\&\, \pi$ then $\pi \in \Pi$.

Therefore, by **st-Val**,

$$\lambda x.e' \in [\![\Theta \vdash x{:}\sigma' \xrightarrow{(\Xi,\Pi)} \tau' \,\&\, (\{\varepsilon\}, \varnothing)]\!]$$

Case the last rule is App

We have $e = x\,y$ and $\Phi = (\{\mathbf{step}\}, \varnothing) \cdot (\Xi, \Pi)$ where

$$\frac{\Delta, \Theta \vdash x : z{:}\tau_1 \xrightarrow{(\Xi,\Pi)} \tau \,\&\, \Phi_1 \quad \Delta, \Theta \vdash y : \tau_1 \,\&\, \Phi_2}{\Delta, \Theta \vdash x\,y : \tau[y/z] \,\&\, (\{\mathbf{step}\}, \varnothing) \cdot (\Xi, \Pi)}$$

By induction hypothesis, we have that

- $x \in [\![\Theta \vdash z{:}\tau_1 \xrightarrow{(\Xi,\Pi)} \tau \,\&\, \Phi_1]\!]$
- $y \in [\![\Theta \vdash \tau_1 \,\&\, \Phi_2]\!]$

Suppose $\rho \vDash \Theta$. Then, $\rho(x) \in [\![z{:}\rho(\tau_1) \xrightarrow{(\Xi,\Pi)} \rho(\tau)]\!]$ and $\rho(y) \in [\![\rho(\tau_1)]\!]$. Let $\rho(x) = \lambda z.e'$. Then,

$$e' \in [\![z{:}\rho(\tau_1) \vdash \rho(\tau) \,\&\, (\Xi, \Pi)]\!]$$

Therefore, the following holds.

- If $e'[\rho(y)/z] \Downarrow v \,\&\, \varpi$ then $v \in \rho(\tau)[\rho(y)/z]$ and $\varpi \in \Xi$.
- If $e'[\rho(y)/z] \Uparrow \bot \,\&\, \pi$ then $\pi \in \Pi$.

Therefore, by **st-App**, **snt-App**, and the definition of trace set concatenation, it follows that

$$x\,y \in [\![\Theta \vdash \tau[y/z] \,\&\, (\{\mathbf{step}\}, \varnothing) \cdot \Phi']\!]$$

Case the last rule is If

We have $e = \texttt{if}\; x \;\texttt{then}\; e_1 \;\texttt{else}\; e_2$ and $\Phi = \Phi_1 \cup \Phi_2$ where

$$\frac{\begin{array}{c} \Delta, \Theta, x = \mathsf{true} \vdash e_1 : \tau \,\&\, (\Xi_1, \Pi_1) \\ \Delta, \Theta, x = \mathsf{false} \vdash e_2 : \tau \,\&\, (\Xi_2, \Pi_2) \end{array}}{\Delta, \Theta \vdash \texttt{if}\; x \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 : \tau \,\&\, (\Xi_1, \Pi_1) \cup (\Xi_2, \Pi_2)}$$

By induction hypothesis, we have that

- $e_1 \in [\![\Theta, x = \mathsf{true} \vdash \tau \,\&\, \Phi_1]\!]$
- $e_2 \in [\![\Theta, x = \mathsf{false} \vdash \tau \,\&\, \Phi_2]\!]$

Suppose $\rho \vDash \Theta$. Then, by simple typing of x, either $\rho(x) = \mathsf{true}$ or $\rho(x) = \mathsf{false}$. Suppose $\rho(x) = \mathsf{true}$. (The case $\rho(x) = \mathsf{false}$ is analogous.) Then, $\rho \vDash \Theta, x = \mathsf{true}$. Therefore, the following holds from $e_1 \in [\![\Theta, x = \mathsf{true} \vdash \tau \,\&\, \Phi_1]\!]$.

- If $\rho(e_1) \Downarrow v \,\&\, \varpi$ then $v \in \rho(\tau)[\rho(y)/z]$ and $\varpi \in \Xi_1$.
- If $\rho(e_1) \Uparrow \bot \,\&\, \pi$ then $\pi \in \Xi_1$.

Therefore, by **st-If1**, **st-If2**, **snt-If1**, and **snt-If2**,

$$\texttt{if}\; x \;\texttt{then}\; e_1 \;\texttt{else}\; e_2 \in [\![\Theta \vdash \tau \,\&\, (\Xi_1, \Pi_1) \cup (\Xi_2, \Pi_2)]\!]$$

Case the last rule is Unreach

We have

$$\frac{\Delta, \Theta \vdash e : \tau \,\&\, \Phi \quad \vDash [\![\Delta, \Theta]\!]_{base} \Rightarrow \bot}{\Delta, \Theta \vdash e : \tau \,\&\, (\varnothing, \varnothing)}$$

It must be the case that $\vDash [\![\Theta]\!]_{base} \Rightarrow \bot$. Therefore, for all ρ, $\neg\rho \vDash \Theta$. Therefore, $e \in [\![\Theta \vdash \tau \,\&\, (\varnothing, \varnothing)]\!]$ holds vacuously.

Case the last rule is Let

Analogous to **Lam** and **App**.

Case the last rule is Event

We have $e = \texttt{ev}[\underline{\mathbf{a}}]$ where

$$\frac{}{\Delta, \Theta \vdash \texttt{ev}[\underline{\mathbf{a}}] : \texttt{unit} \,\&\, (\{a\}, \varnothing)}$$

Suppose $\rho \vDash \Theta$. Then, $\rho(\texttt{ev}[\underline{\mathbf{a}}]) = \texttt{ev}[\underline{\mathbf{a}}]$ and by **st-Ev**, $\texttt{ev}[\underline{\mathbf{a}}] \Downarrow () \,\&\, a$. Therefore, by **Const** and **Sub**, $\texttt{ev}[\underline{\mathbf{a}}] \in [\![\Theta \vdash \texttt{unit} \,\&\, (\{a\}, \varnothing)]\!]$.

$\square$