A Fixpoint Logic and Dependent Effects for Temporal Property Verification

Yoji Nanjo
University of Tsukuba
nanjo@logic.cs.tsukuba.ac.jp

Hiroshi Unno
University of Tsukuba / RIKEN AIP
uhiro@cs.tsukuba.ac.jp

Eric Koskinen
Stevens Institute of Technology
eric.koskinen@stevens.edu

Tachio Terauchi
Waseda University
terauchi@waseda.jp

Abstract

Existing approaches to temporal verification of higher-order functional programs have either sacrificed compositionality in favor of achieving automation or vice-versa. In this paper we present a dependent-refinement type & effect system to ensure that well-typed programs satisfy given temporal properties, and also give an algorithmic approach—based on deductive reasoning over a fixpoint logic—to typing in this system. The first contribution is a novel type-and-effect system capable of expressing *dependent temporal* effects, which are fixpoint logic predicates on event sequences and program values, extending beyond the (non-dependent) temporal effects used in recent proposals. Temporal effects facilitate compositional reasoning whereby the temporal behavior of program parts are summarized as effects and combined to form those of the larger parts. As a second contribution, we show that type checking and typability for the type system can be reduced to solving first-order fixpoint logic constraints. Finally, we present a novel deductive system for solving such constraints. The deductive system consists of rules for reasoning via invariants and well-founded relations, and is able to reduce formulas containing both least and greatest fixpoints to predicate-based reasoning.

CCS Concepts • **Theory of computation → Programming logic**; **Program verification**; • **Software and its engineering → Formal software verification**;

Keywords higher-order programs, temporal verification, fixpoint logic, dependent temporal effects, dependent refinement types

ACM Reference Format:
Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209204

1 Introduction

Recent years have seen many new approaches for verifying temporal properties of higher-order programs. At first, these works were restricted to safety properties [9, 20, 23–25], termination [13, 28], non-termination [14], or finite data [18]. Algorithmic reductions based on higher-order recursion schemes [7, 8] and constrained Horn clause solving [3] have enjoyed automation success. Other works have shown that the automata-theoretic reduction to fair-termination [27] can be lifted to the higher-order setting [16]. Still other works have permitted reasoning about angelic-vs-demonic nondeterminism [26].

Meanwhile there has been a sub-community, whose aim is to support temporal specifications directly in the type system, in the form of temporal *effects*. The promise of this approach is that it may lead to a more compositional verification strategy, where temporal reasoning can be done locally (at the level of terms, expressions, functions, etc.) and combined together via an orchestrating type system to reason about the overall program [4, 12, 21]. These works, however, required an over-approximation to cope with the effects of recursive functions. In particular, the temporal effects in prior work are simply sets of event traces that coarsely over-approximate the actual temporal behavior of the program terms either via ω-regular sets [4] or else by allowing recursive functions to have any infinite effect [12]. These treatments preclude specifying value-dependent temporal properties as effects, and also, for infinite-state programs, the over-approximation may result in loss of precision even when the goal property to be verified is non-dependent.

In summary, while recent works have led to advanced *non-compositional* algorithmic approaches, the state-of-the-art is that we don't have a clear theory to connect compositional type & effect-based approaches with algorithmic verification techniques. Bridging this gap could mean exploiting the best of both worlds.

In this paper, we bridge this gap, presenting methods for algorithmic verification of temporal properties specified as effects. Our first step is to raise the bar a little higher. We introduce the concept of *dependent* temporal effects. Our types have the form $(\tau \mathbin{\&} (\Phi^\mu, \Phi^\nu))$ where we use dependent-refinement types and, as in prior work [4, 12, 21], the effects are a pair: Φ^μ corresponding to the finite effects and Φ^ν corresponding to the infinite effects. Unlike prior work, we treat these (finite and infinite) effects of program expressions as *predicates* on finite and infinite (respectively) event sequences–*i.e.*, a predicate on Σ^* and a predicate on Σ^ω–over some alphabet of events Σ. As discussed below, the predicates are also on program values, thus making the effects value-dependent. Moreover, we express these predicates in a fixpoint logic that permits least- and greatest-fixpoints of predicate variables and has base theories of integers and finite/infinite event sequences. We can express, for example, that the effect of a function `foo n` is given by

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209204

the pair $(\Phi^{\mu}_{\texttt{foo}}, \Phi^{\nu}_{\texttt{foo}})$ defined as:

$$\Phi^{\mu}_{\texttt{foo}} \triangleq \lambda x.\bot \qquad \Phi^{\nu}_{\texttt{foo}} \triangleq \lambda x.x \in ((\underline{\textbf{Ready}} \cdot \underline{\textbf{Send}}^{\texttt{n}}) \mid \underline{\textbf{Wait}})^{\omega}$$

The effect predicate Φ^{μ} specifies that there are no finite effects whereas Φ^{ν} specifies that the infinite behavior is to repeatedly generate either (i) a **Ready** event and n **Send** events or (ii) a single **Wait** event. Notice, in particular, that n is a parameter to foo, making this effect *dependent* with respect to foo's argument.

Next, we provide dependent temporal effect *typing rules*, which relate the effects of one program part to the effects of others, accumulating proof obligations in the form of constraints along the way. The recursive function definition rule highlights our treatment, as well as the benefit of treating effects as finite/infinite predicates. In prior work, over-approximations of effects were used. Here we instead relate the effect Φ of the body of the function e with the effect of the overall recursive function $\texttt{rec}(f, \widetilde{x}, e)$ with two constraints: a least fixpoint constraint relating finite effects Φ^{μ} to the finite effects of $\texttt{rec}(f, \widetilde{x}, e)$ and a greatest fixpoint constraint relating the infinite effects Φ^{ν} to the infinite effects of $\texttt{rec}(f, \widetilde{x}, e)$. These effects of recursive functions have the form:

$$\Phi^{\mu}_{\texttt{foo}} = \lambda x.(\mu X_{\mu}(n, x).\ldots)(n, x) \quad \Phi^{\nu}_{\texttt{foo}} = \lambda x.(\nu X_{\nu}(n, x).\ldots)(n, x)$$

where X_{μ} and X_{ν} are *effect predicate variables* (cf. Sec.4.1). Our treatment of effects as predicates is key to enabling an overall type system that is able to remain precise, even in the context of representing infinite behaviors. In our type system, constraints are also imposed, for example, in instances of subtyping.

The question then remains: how do we solve these constraints? Addressing this question leads to the next contribution of our work, which achieves a marriage between type-and-effect-based temporal specifications [4, 12, 21] and algorithmic verification approaches [8, 9, 13, 16, 20, 22–24]. We introduce a deductive system for reasoning about these fixpoint constraints. The deductive proof rules let us address least- and greatest-fixpoint constraints that appear in the typing tree. The rules reduce the fixpoint subformula to reasoning about invariants and well-founded relations. The use of invariants and well-founded relations is motivated by their use in safety and liveness verification of infinite state programs (as mentioned above), and enables solving constraints that cannot be solved by a simple unrolling of the fixpoint formula. Also, from an engineering point of view, one can leverage existing tools to synthesize invariants and well-founded relations. The particular strategy we employ depends on the kind of fixpoint (least or greatest) and whether they occur in negative or positive position in the formula. Our deductive system then has a collection of further approximation rules, defined inductively on the structure of the formula, that further reduce the formulas to predicate-based reasoning.

Contributions. In summary, we make the following contributions:

1. *Dependent* temporal effects, expressed in a first-order fixpoint logic over theories of integers and finite/infinite event sequences, wherein those integers can depend on program values. (Sec. 3)
2. A type system for dependent temporal effects, supporting programs written in an ML-like language with higher-order features and ranging over infinite data. (Sec. 4.4)
3. A soundness proof of our type system. (Theorem 4.1)
4. A deductive proof system that employs invariants and well-founded relations to solve formulas in the fixpoint logic containing both least and greatest fixpoints. (Sec. 5)
5. A soundness proof for our deductive rules. (Theorem 5.2)

Organization. In the next section, we give an example and use it to highlight our main contributions, as well as some further examples to show the applicability of our work. In Sec. 3 we give our ML-like language and in Sec. 4 we present our type system and associated soundness theorem. Our deductive fixpoint proof system is given in Sec. 5. We conclude with a discussion of related work in Sec. 6. Omitted materials appear in the extended technical report [17].

2 Overview

We now give a summary of our techniques, using the example shown in Fig. 1. At the end of this section we provide further examples (Fig. 2) that illustrate the applicability of our work.

Our type and effect system, extended with *dependent* temporal effects can be illustrated with this messenger example. This example simulates a client interacting with a server. The messenger program calls until_ready which will make a nondeterministic boolean choice: in one case it will trigger the event **Ready** and otherwise it will **Wait** and again call until_ready. If the **Ready** event ever occurs, then until_ready will return, and send_msgs will generate n instances of **Send**. Finally, messenger will recur. This program has no finite traces. Its infinite traces follow the form of the *dependent* ω-regular expression $((\underline{\textbf{Ready}} \cdot \underline{\textbf{Send}}^{\texttt{n}}) \mid \underline{\textbf{Wait}})^{\omega}$. Notice that this effect depends on the input to the program n. Although this example is simple, it already illustrates a property that cannot be expressed in prior work [3, 9, 13, 14, 16, 20, 23–26, 28]. In fact, this effect expression escapes classical LTL or the μ-calculus.

We now discuss how our approach can conclude the above dependent ω-regular expression, highlighting the contributions along the way. The typings for the recursive procedures in this example can be found in Fig. 1. (The full type derivation for send_msgs is also shown, and the type derivation for until_ready is given in [17].) The overall type for send_msgs is $\tau_{\texttt{send\_msgs}} = (n : \{n \mid n \geq 0\}) \to (\texttt{unit} \,\&\, \Phi_{\texttt{send\_msgs}})$. We assume that the reader is already familiar with dependent-refinement types such as above, which states that send_msgs is a function from non-negative integers to unit, having effects described by $\Phi_{\texttt{send\_msgs}}$. As in prior work [4, 12], effects are given as pairs, i.e., $\Phi_{\texttt{send\_msgs}} = (\Phi^{\mu}_{\texttt{send\_msgs}}, \Phi^{\nu}_{\texttt{send\_msgs}})$ the first corresponding to the finite effects of send_msgs and the latter to the infinite effects.

In this paper, we introduce *dependent* temporal effects. To this end, we begin by treating each component's effect as *predicates*. For send_msgs, we have:

$$\begin{aligned}
\Phi^{\mu}_{\texttt{send_msgs}} = \quad & \lambda x.(\mu X_{\mu}(n, x).(n = 0 \land x = \epsilon \\
& \lor n \neq 0 \land \exists y.x = \underline{\textbf{Send}} \cdot y \land X_{\mu}(n-1, y)))(n, x) \\
\Phi^{\nu}_{\texttt{send_msgs}} = \quad & \lambda x.(\nu X_{\nu}(n, x). \\
& n \neq 0 \land \exists y.x = \underline{\textbf{Send}} \cdot y \land X_{\nu}(n-1, y))(n, x)
\end{aligned}$$

As discussed later, our type system is able to derive this judgment from the syntax of the program. The first component $\Phi^{\mu}_{\texttt{send\_msgs}}$ describes the effects of the finite traces of send_msgs via a predicate $\lambda x.\_$ where x will be a candidate event sequence. The body is a least fixpoint equation over a *predicate variable* X_{μ}, parameterized by variables n and x. The fixpoint's body has two cases: when $n = 0$, then the event sequence is simply empty, denoted ϵ. Otherwise,

(a) Source Code

```
let rec until_ready () =
  if * then
    (event[Ready]; ())
  else
    (event[Wait];
     until_ready ())

let rec send_msgs n =
  if n = 0 then ()
  else
    (event[Send];
     send_msgs (n-1))

let rec messenger n =
  until_ready ();
  send_msgs n;
  messenger n
```

(b) Typing Rules and Final Effect Approximations

Types

$$\begin{aligned}
\tau_{\texttt{until_ready}} &= \texttt{unit} \to (\texttt{unit} \mathbin{\&} \Phi_{\texttt{until_ready}})\\
\Phi^{\mu}_{\texttt{until_ready}} &= \lambda x.(\mu X_\mu(x).x = \underline{\mathbf{Ready}} \vee \exists y.x = \underline{\mathbf{Wait}} \cdot y \wedge X_\mu(y))(x)\\
\Phi^{\nu}_{\texttt{until_ready}} &= \lambda x.(\nu X_\nu(x).x = \underline{\mathbf{Wait}} \cdot y \wedge X_\nu(y))(x)\\
\tau_{\texttt{send_msgs}} &= (n : \{n \mid n \geq 0\}) \to (\texttt{unit} \mathbin{\&} \Phi_{\texttt{send_msgs}})\\
\Phi^{\mu}_{\texttt{send_msgs}} &= \lambda x.(\mu X_\mu(n,x).(n = 0 \wedge x = \epsilon \vee n \neq 0 \wedge \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\mu(n-1,y)))(n,x)\\
\Phi^{\nu}_{\texttt{send_msgs}} &= \lambda x.(\nu X_\nu(n,x).n \neq 0 \wedge \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\nu(n-1,y))(n,x)\\
\tau_{\texttt{messenger}} &= (n : \{n \mid n \geq 0\}) \to (\texttt{unit} \mathbin{\&} \Phi_{\texttt{messenger}})\\
\Phi^{\mu}_{\texttt{messenger}} &= \lambda x.(\mu X_\mu(n,x).\exists y_1,y_2.x = y_1 \cdot y_2 \wedge (\Phi'_{\texttt{until_ready}} \cdot \Phi'_{\texttt{send_msgs}})^{\mu}(y_1) \wedge X_\mu(n,y_2))(n,x)\\
\Phi^{\nu}_{\texttt{messenger}} &= \lambda x.(\nu X_\nu(n,x).\left(\begin{array}{l}(\Phi'_{\texttt{until_ready}} \cdot \Phi'_{\texttt{send_msgs}})^{\nu}(x) \vee\\ \exists y_1,y_2.x = y_1 \cdot y_2 \wedge\\ \quad (\Phi'_{\texttt{until_ready}} \cdot \Phi'_{\texttt{send_msgs}})^{\mu}(y_1) \wedge X_\nu(n,y_2)\end{array}\right))(n,x)
\end{aligned}$$

Final Effect Approximations

$$\begin{aligned}
\Phi'_{\texttt{until_ready}} &= (\lambda x.x \in \underline{\mathbf{Wait}}^{*} \cdot \underline{\mathbf{Ready}}, \lambda x.x \in \underline{\mathbf{Wait}}^{\omega})\\
\Phi'_{\texttt{send_msgs}} &= (\lambda x.x \in \underline{\mathbf{Send}}^{n}, \lambda x.\bot)\\
\Phi'_{\texttt{messenger}} &= (\lambda x.\bot, \lambda x.x \in (\underline{\mathbf{Ready}} \cdot \underline{\mathbf{Send}}^{n} \mid \underline{\mathbf{Wait}})^{\omega})
\end{aligned}$$

(c) Type Derivation Tree for `send_msgs`, including Deductive Fixpoint Rules ($\Vdash$)

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\texttt{send\_msgs} : \tau, n = 0 \vdash () : \sigma_{\texttt{if}}} \quad \dfrac{\vdots}{\texttt{send\_msgs} : \tau, n > 0 \vdash \texttt{ev}[\underline{\mathbf{Send}}];\texttt{send\_msgs}\ (n-1) : \sigma_{\texttt{if}}}}{\texttt{send\_msgs} : \tau, n \geq 0 \vdash \begin{array}{l}\texttt{if}\ n = 0\ \texttt{then}\ ()\\ \texttt{else}\ (\texttt{ev}[\underline{\mathbf{Send}}];\texttt{send\_msgs}\ (n-1))\end{array} : \sigma_{\texttt{if}}}}{\vdash \texttt{rec}(\texttt{send\_msgs}, n, \cdots) : (n : n \geq 0) \to (\texttt{unit} \mathbin{\&} \Phi_{\texttt{send\_msgs}})} \quad \dfrac{\dfrac{\mathbf{A} \quad \mathbf{B}}{n \geq 0 \vdash \left(\begin{array}{l}(\texttt{unit} \mathbin{\&} \Phi_{\texttt{send\_msgs}})\\ <:(\texttt{unit} \mathbin{\&} \Phi'_{\texttt{send\_msgs}})\end{array}\right)}}{\vdots}}{\vdash \texttt{rec}(\texttt{send\_msgs}, n, \cdots) : (n : n \geq 0) \to (\texttt{unit} \mathbin{\&} \Phi'_{\texttt{send\_msgs}})}$$

$$\tau \triangleq (n : n \geq 0) \to (\texttt{unit} \mathbin{\&} (\lambda x.X_\mu(n,x), \lambda x.X_\nu(n,x)))$$

$$\sigma_{\texttt{if}} \triangleq (\texttt{unit} \mathbin{\&} (\lambda x.n = 0 \wedge x = \epsilon \vee n \neq 0 \wedge \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\mu(n-1,y), \lambda x.\exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\nu(n-1,y)))$$

A :

$$\dfrac{\models (n = 0 \wedge x = \epsilon \vee \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge y \in \underline{\mathbf{Send}}^{n-1}) \Rightarrow x \in \underline{\mathbf{Send}}^{n} \quad \dfrac{\models n \geq 0 \Rightarrow (x \in \underline{\mathbf{Send}}^{n} \Rightarrow x \in \underline{\mathbf{Send}}^{n})}{\Vdash n \geq 0 \Rightarrow (x \in \underline{\mathbf{Send}}^{n} \Rightarrow x \in \underline{\mathbf{Send}}^{n})}}{\Vdash n \geq 0 \Rightarrow (\Phi^{\mu}_{\texttt{send\_msgs}}(x) \Rightarrow \Phi'^{\mu}_{\texttt{send\_msgs}}(x))}$$

B :

$$\dfrac{\dfrac{\dfrac{\models (p_1(n,x) \wedge n = 0) \Rightarrow \neg(n \neq 0)}{X_\nu(n,x);p_1;p_2;n = 0 \mathrel{\overline{\uparrow}} n \neq 0} \quad \dfrac{\dfrac{\dfrac{\models (p_1(n,x) \wedge n \neq 0 \wedge x \neq \underline{\mathbf{Send}} \cdot x') \Rightarrow \neg(x = \underline{\mathbf{Send}} \cdot x')}{X_\nu(n,x);p_1;p_2;n \neq 0 \wedge x \neq \underline{\mathbf{Send}} \cdot x' \mathrel{\overline{\uparrow}} x = \underline{\mathbf{Send}} \cdot x'} \quad \dfrac{\begin{array}{l}\models \quad (p_1(n,x) \wedge n \neq 0 \wedge x = \underline{\mathbf{Send}} \cdot x')\\ \qquad \Rightarrow (p_1(n-1,x') \wedge p_2(n,x,n-1,x'))\end{array}}{X_\nu(n,x);p_1;p_2;n \neq 0 \wedge x = \underline{\mathbf{Send}} \cdot x' \mathrel{\overline{\uparrow}} X_\nu(n-1,x')}}{X_\nu(n,x);p_1;p_2;n \neq 0 \mathrel{\overline{\uparrow}} x = \underline{\mathbf{Send}} \cdot x' \wedge X_\nu(n-1,x')}}{X_\nu(n,x);p_1;p_2;n \neq 0 \mathrel{\overline{\uparrow}} \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\nu(n-1,y)}}{X_\nu(n,x);p_1;p_2;\top \mathrel{\overline{\uparrow}} n \neq 0 \wedge \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\nu(n-1,y)} \quad \dfrac{\models n \geq 0 \Rightarrow (\neg(n \geq 0) \Rightarrow \bot)}{\Vdash n \geq 0 \Rightarrow (\neg(n \geq 0) \Rightarrow \bot)}}{\Vdash n \geq 0 \Rightarrow (\Phi^{\nu}_{\texttt{send\_msgs}}(x) \Rightarrow \Phi'^{\nu}_{\texttt{send\_msgs}}(x))}$$

$$p_1 \triangleq \lambda(n,x).n \geq 0 \qquad p_2 \triangleq \lambda(n_1,x_1,n_2,x_2).n_1 > n_2 \geq 0$$

Figure 1. *Clockwise*: (a) Source code for `messenger`; (b) Types & effects for recursive functions along with our final effect conclusions; and (c) type derivation for `send_msgs`, including the use of our deductive proof rules ($\Vdash$) in subtrees **A** and **B**.

the predicate specifies that x will be the $\underline{\mathbf{Send}}$ event, followed by some event sequence y and that $X_\mu(n-1,y)$ must hold. Overall, this fixpoint is applied to variables n and x.

The second component $\Phi^{\nu}_{\texttt{send\_msgs}}$ describes the infinite effects of `send_msgs`. Not surprisingly, a greatest fixpoint equation is used, with predicate variable X_ν again parameterized by n and x. The $n = 0$ case is finite and not possible. We will see momentarily that the other infinite case is also not possible.

Amortized Complexity

```
let rev l =
  let rec aux l acc = match l with
    | [] -> acc | h::t ->
       event[Tick]; aux t (h::acc)
  in aux l []
let is_empty (l1,l2) = l1 = [] && l2 = []
let enqueue e (l1,l2) = event[Enq];(l1,e::l2)
let rec dequeue (l1,l2) = match l1 with
  | [] -> dequeue (rev l2, [])
  | e::l1' -> event[Deq]; (e, (l1', l2))
let rec main (l1,l2) =
  if * then main (enqueue 42 (l1,l2))
  else if is_empty (l1,l2) then ()
  else main (snd (dequeue (l1,l2)))
```

$\texttt{main} : ((l1, l2) : \texttt{int list} \times \texttt{int list}) \rightarrow (\texttt{unit} \,\&\, \Phi)$
$\Phi^\mu = \lambda x.\#_{\underline{\texttt{Enq}}}(x) + |l2| = \#_{\underline{\texttt{Tick}}}(x) = \#_{\underline{\texttt{Deq}}}(x) - |l1|$
$\Phi^\nu = \lambda x.\top$

Higher-Order

```
let rec zoom () =
  event[Zoom]; zoom ()

let rec shrink t f d =
  if f () <= 0 then
    zoom ()
  else
    (event[Shrink];
     let t' = f() - d in
     shrink t' (fun x -> t') d)

let shrinker t d =
  shrink t (fun x -> t) d
```

$\texttt{shrinker} : (t : \{t \mid t \geq 0\}) \rightarrow (d : \{d \mid d > 0 \land t \bmod d = 0\}) \rightarrow (\texttt{unit} \,\&\, \Phi)$
$\Phi^\mu = \lambda x.\bot$
$\Phi^\nu = \lambda x.x \in \underline{\texttt{Shrink}}^{t/d} \cdot \underline{\texttt{Zoom}}^\omega$

Web Server Fairness

```
let rec listener npool pend =
 if * && pend < npool then
   (event[Accept];
    listener npool (pend + 1))
 else if pend > 0 then
   (event[Handle];
    listener npool (pend - 1))
 else
   (event[Wait];
    listener npool pend)

let server npool =
  listener npool 0
```

$\texttt{server} : (\texttt{npool} : \{\nu \mid \nu \geq 0\}) \rightarrow (\texttt{unit} \,\&\, (\lambda x.\bot, \lambda x.\phi))$
$\phi = \begin{pmatrix} x \in (\Sigma^* \cdot (\Sigma \setminus \underline{\texttt{Accept}})^{\texttt{npool}+1})^\omega \\ \Rightarrow x \in (\Sigma^* \cdot \underline{\texttt{Wait}})^\omega \end{pmatrix}$

Figure 2. Further examples of programs and corresponding dependent temporal effects that we are able to verifying using our approach.

Our typing judgments impose proof obligations in the form of constraints. Most notably, the type rule for recursive function definition (cf. T-Fun in Sec. 4) for a function f requires that the effect of a total application of f be compatible with the effect of the body of f, which is itself derived from the typing rules. Roughly, T-Fun works as follows. First, it checks that the body of f has finite/infinite effect pair (Φ^μ, Φ^ν), under a typing environment where a total application of f has some finite/infinite effect pair $(\lambda x \in \Sigma^*.X_\mu(\widetilde{x}, x), \lambda x \in \Sigma^\omega.X_\nu(\widetilde{x}, x))$. X_μ and X_ν are finite and infinite predicate variables, respectively. Given this, the effect of a total application of the recursive function is then the effect pair $(\lambda x \in \Sigma^*.q_\mu(\widetilde{x}, x), \lambda x \in \Sigma^\omega.q_\nu(\widetilde{x}, x))$ where our type system requires that $q_\mu = \mu X_\mu(\widetilde{x}, x).\Phi^\mu(x)$ and $q_\nu = \nu X_\nu(\widetilde{x}, x).[q_\mu/X_\mu]\Phi^\mu(x)$. In this way, we require that the finite (resp., infinite) effects of the recursive function be given by a least (resp., greatest) fixpoint over a predicate variable X_μ (resp., X_ν). The type system also generates constraints in other rules, such as the subtyping rules (S-Qual, etc.) of the form $(\tau_1 \,\&\, \Phi_1) <: (\tau_2 \,\&\, \Phi_2)$. In these cases, the type system requires that the finite (resp., infinite) effects Φ_1^μ (resp., Φ_1^ν) *is approximated by* the finite (resp., infinite) effects Φ_2^μ (resp., Φ_2^ν).

Addressing the recursive function rule in the type soundness proof is a challenge due to the infinite effects. We use a semantics of types and an infinite sequence of approximations for the recursive function and its infinite effect. This infinite sequence of approximations is used to construct the greatest fixpoint.

The types for `messenger` are given in the second column of Fig. 1. Let us consider the `send_msgs` recursive function, whose overall type is given by the dependent-refinement type $\tau_{\texttt{send\_msgs}}$, that constrains input n to be greater than or equal to 0. The overall effect $\Phi_{\texttt{send\_msgs}}$ has two parts: the finite effect $\Phi^\mu_{\texttt{send\_msgs}}$ and the infinite effect $\Phi^\nu_{\texttt{send\_msgs}}$. These effect predicates involve predicate *variables* X_μ and X_ν, quantified with a least and greatest fixpoint, respectively. Notice that X_μ and X_ν are parameterized by n, which is a *program* variable: the input to `messenger`. This highlights our support for *dependent* temporal effects, showing how they are treated intimately with the fixpoint constraints on recursive functions.

Solving Fixpoints via Our Deductive Proof Rules. The deductive rules enable us to conclude the final effects:

$$\begin{aligned} \Phi'_{\texttt{until\_ready}} &= (\lambda x.x \in \underline{\texttt{Wait}}^* \cdot \underline{\texttt{Ready}}, \lambda x.x \in \underline{\texttt{Wait}}^\omega) \\ \Phi'_{\texttt{send\_msgs}} &= (\lambda x.x \in \underline{\texttt{Send}}^n, \lambda x.\bot) \\ \Phi'_{\texttt{messenger}} &= (\lambda x.\bot, \lambda x.x \in (\underline{\texttt{Ready}} \cdot \underline{\texttt{Send}}^n \mid \underline{\texttt{Wait}})^\omega) \end{aligned}$$

Intuitively, `until_ready` has finite behaviors that repeat **Wait** finitely many times followed by **Ready**. The infinite behaviors of `until_ready` are infinite repetition of **Wait**. `send_msgs` has only finite behaviors, specifically, repetition of **Send** n times, where n is the input to the overall program `messenger`. Finally, `messenger` has only infinite effects, that arises from a combination of the other two functions. Notice that our approach follows the classical compositional spirit of type systems: conclusions about terms are derived independently and then combined together to construct conclusions about compound terms. Similarly, our conclusion about the dependent temporal effects of method `messenger` is constructed *after* we have reached conclusions about the effects of its callees (including approximations of these callees).

So, how do we come to these final approximations of all functions? Our sub-typing rules (cf. Fig. 7 in Sec. 4) allow us to introduce an approximation effect predicate Φ' of effect predicate Φ provided that we can show that $\forall x.\Phi'(x) \Rightarrow \Phi(x)$. For `send_msgs`, the subtyping appears in Fig. 1, with premises **A** and **B**.

Our deductive system comprises rules for reasoning about these formulas that contained least- and greatest- fixpoint formulas buried within them. The key idea is to reduce these tricky subformulas to invariants and well-founded relations, both described as predicates, and then symbolically manipulate the side-conditions that arise until they can be handled by base solvers. The process begins with one of four main rules, under- or over-approximate (as the case may be) least and greatest fixpoints, depending on whether they appear in a negative or positive position in the fixpoint formula. We'll now look at how two of the rules can be used for the example.

First, looking at the *finite* effects of `send_msgs`, our rules allow us to show, for example, that $\lambda x.x \in \underline{\mathbf{Send}}^n$ approximates $\Phi^{\mu}_{\texttt{send\_msgs}}$ by using invariant predicates that over-approximate the least fixpoints. This can be seen in the deductive proof rule in subtree **A**. We have a formula, where the least fixpoint occurs in a negative position, i.e., inside $\Phi^{\mu}_{\texttt{send\_msgs}}$. Our proof rule (Fp-Lfp$^-$) lets us approximate this buried least fixpoint with $\lambda(n,x).x \in \underline{\mathbf{Send}}^n$ by using the *pre*-fixpoint. In the first premise of the rule, we consider only the fixpoint and must show that when we substitute $\lambda(n,x).x \in \underline{\mathbf{Send}}^n$ into the fixpoint formula, the result is approximated by $\lambda(n,x).x \in \underline{\mathbf{Send}}^n$. In the second premise, we use this information, eliminating the fixpoint.

Next, looking at the *infinite* effects of `send_msgs`, our proof system lets us show that the goal effect approximation $\lambda x.\bot$ of $\Phi^{\nu}_{\texttt{send\_msgs}}$ holds. This is done by, first, over-approximating the greatest fixpoint subformula that occurs in a negative position using a predicate and a well-foundedness check. Note that the typing judgments accumulate the invariant that $n \geq 0$, and that it is incorporated into the deductive proof rule in subtree **B**. The rule (Fp-Gfp$^-$) lets us replace the GFP formula $(\nu X(\widetilde{x}).\psi)(\widetilde{t})$ by some predicate $\neg p_1(\widetilde{t})$. There is a side condition, however, that we must also provide a relational well-foundedness predicate p_2 which witnesses that $\neg p_1$ over-approximates $\nu X(\widetilde{x}).\top \wedge \psi$. In the `send_msgs` example, we use the predicate $\neg(n \geq 0)$ to approximate the GFP formula in $\Phi^{\nu}_{\texttt{send\_msgs}}$. What remains is the side-condition, where we use $p_2 = n_1 > n_2 \geq 0$ to witness that $\neg p_1$ over-approximates $n \neq 0 \wedge \exists y.x = \underline{\mathbf{Send}} \cdot y \wedge X_\nu(n-1, y)$.

We treat this side-condition of witnessing predicates' approximations itself as a judgment (denoted $X(\widetilde{x}); p_1; p_2; \top \mathrel{\bar{\uparrow}} \psi$) as well as an analogous least-fixpoint judgment (denoted $X(\widetilde{x}); p_1; p_2; \bot \downarrow \psi$) in another series of proof rules. These rules are inductively defined over ψ, letting us discharge this obligation syntactically down to predicate reasoning, as can be seen in the rest of proof subtree **B**. Specifically, we use a rule for conjunction (Apx$^\nu$-$\wedge$), existential quantification (Apx$^\nu$-$\exists$), and then conjunction again. Each rule has premises for each sub-formula(e) and predicate-oriented side conditions. We will discuss these rules in Sec. 5.

Other Examples & Applications. The `messenger` example is intended to be a small example that highlights some of the main aspects of our work. In Fig. 2 we provide the source code and effect-based temporal properties for more examples, demonstrating the applicability of our approach. (The types for these examples are given in [17].) We now discuss each program.

Amortized Complexity. This example involves functions that manipulate a pair of integer lists. The `main` loop will nondeterministically enqueue a new integer, via `enqueue` which adds the element to the `l2` list. If `main` finds that the list is empty, it terminates. Otherwise, it iterates, but only after applying `dequeue` to the list. `dequeue` shuffles elements between `l1` and `l2`: if `l1` is empty, it moves everything from `l2` to `l1` and, otherwise, it dequeues by returning a pair of the dequeued item and the new queue `(l1',l2)`. Here, $\#\underline{\mathbf{a}}(x)$ is the number of $\underline{\mathbf{a}}$'s in x. The temporal effect Φ of `main` asserts that, when the program terminates, the number of enqueues plus the length of `l2` is equal to the number of dequeues minus the length of `l1`, which is equal to the number of $\underline{\mathbf{Tick}}$'s.

$$
\begin{array}{rl}
\text{(events)} & \underline{\mathbf{a}} ::\in \Sigma \\
\text{(expressions)} & e ::= x \mid n \mid v_1\ op\ v_2 \mid \texttt{rec}(f,\widetilde{x},e) \mid v_1\ v_2 \mid \texttt{ev}[\underline{\mathbf{a}}] \\
& \quad \mid \texttt{ifz}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x\texttt{=}\ e_1\ \texttt{in}\ e_2 \\
\text{(values)} & v ::= x \mid n \mid \texttt{rec}(f,\widetilde{x},e)\,\widetilde{v} \qquad (\text{where } |\widetilde{v}| < |\widetilde{x}|) \\
\text{(simple types)} & T ::= \texttt{int} \mid T_1 \to T_2
\end{array}
$$

Figure 3. Syntax of $\mathcal{L}$

Higher-Order Functions. The second example `shrinker` contains a higher-order function `shrink`. The example is adopted from a similar example in [12]. Here, `shrink` takes an argument `f` which is a function from `unit` to `int`, and an integer argument `d`. Then, it recursively calls itself by passing a function that returns `d` less than the given function, until `f` returns a non-positive value. Here, `t` is a *ghost parameter* that is used to represent sufficient information about the passed function (see., e.g., [25]). The effect Φ asserts that `shrinker` never terminates, and its infinite executions emit the event sequences $\underline{\mathbf{Shrink}}^{\texttt{t/d}} \cdot \underline{\mathbf{Zoom}}^{\omega}$. That is, `shrink` is called `t/d` times, followed by infinitely many calls to `zoom`.

Server Fairness/Liveness. The function `listener` in this example simulates a non-terminating loop within, e.g., a web server, that awaits new incoming connections ($\underline{\mathbf{Wait}}$), accepts them ($\underline{\mathbf{Accept}}$) and dispatches them to an appropriate handler ($\underline{\mathbf{Handle}}$). Argument `pend` is the number of clients that have been accepted but not yet dispatched and argument `npool` is an upper bound on the amount of clients that can be accepted but yet undispatched at a given time. The use of $*$ indicates a non-deterministic boolean choice.

One critical property is that every accepted connection is eventually handled, i.e., that the pool of pending clients eventually becomes empty. This is, however, not true in general since infinitely many new clients may preempt handling pending clients. The property must be instead weakened to include a fairness constraint that all infinite event streams satisfy $(\Sigma^* \cdot (\Sigma \setminus \{\underline{\mathbf{Accept}}\})^{\texttt{npool+1}})^{\omega}$, i.e., that there will always eventually be a time when new connections won't be accepted for `npool` + 1 steps. (Technically, this does ensure that the pool of clients always eventually becomes empty, even though less than `npool` + 1 steps may be needed.)

These examples demonstrate an interesting connection of our method and works that have been focused on resource analysis and cost semantics. One way of thinking about the execution time of a program is by considering the events generated by the program (as we discuss in Sec. 3, we require programs not to have infinite event-less executions). Our dependent temporal effects are capable of expressing specifications of programs that limit the number of events that could possibly be generated, a phenomenon that corresponds to an upper bound on computation time. We believe that there is interesting future work to be explored at the intersection of these two research tracks.

3 Target Language

The syntax of an ML-like (i.e., typed, higher-order, and call-by-value) functional language $\mathcal{L}$ is shown in Fig. 3. Here, n, x, and $\underline{\mathbf{a}}$ are meta-variables ranging respectively over integers, term variables, and events. Σ represents a finite set of events. We write $\widetilde{x}$ for a finite sequence of variables and $|\widetilde{x}|$ for the length of $\widetilde{x}$. We also write ϵ for the empty sequence. We use a meta-variable ϖ (resp.

$$
\begin{array}{rl}
\text{(formulas) } \phi & ::= \top \mid \bot \mid A(\widetilde{t}) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\
& \mid \ \phi_1 \Rightarrow \phi_2 \mid \forall x \in \mathcal{S}.\phi \mid \exists x \in \mathcal{S}.\phi \mid X(\widetilde{t}) \\
& \mid \ (\mu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi)(\widetilde{t}) \mid (\nu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi)(\widetilde{t}) \\
\text{(terms) } t & ::= x \mid f(\widetilde{t}) \qquad \text{(predicates) } p ::= \lambda\widetilde{x} \in \widetilde{\mathcal{S}}.\phi \\
\text{(sorts) } \mathcal{S} & ::\in \{\mathbb{Z}, \Sigma^*, \Sigma^\omega\}
\end{array}
$$

Figure 4. Syntax of fixpoint logic

π) to represent a finite (rep. infinite) sequence of events. We write $\varpi \cdot \pi$ (resp. $\varpi \cdot \varpi'$) for the concatenation of the finite ϖ and the infinite π (resp. finite ϖ') sequences. *op* represents binary integer operators such as $+, -, \times, =$, and $<$. We assume that boolean and unit values are encoded as integers (e.g., $\texttt{true} = 0$ and $\texttt{false} = 1$).

We assume that expressions are simply-typed. An expression $\texttt{ev}[\underline{\mathbf{a}}]$ raises the event $\underline{\mathbf{a}}$. An expression $\texttt{ifz}\,v\,\texttt{then}\,e_1\,\texttt{else}\,e_2$ reduces to e_1 if $v = 0$ and e_2 otherwise. We abbreviate $\texttt{let}\,x{=}e_1\,\texttt{in}\,e_2$ as $e_1; e_2$ if x does not occur in e_2. An expression $\texttt{rec}(f, \widetilde{x}, e)$ represents a (possibly recursive) function f with the arguments $\widetilde{x}$ (where $|\widetilde{x}| \geq 1$) and the body e. We assume that $\texttt{rec}(f, \widetilde{x}, e)$ is productive: if a run of the function is non-terminating, it exhibits an infinite sequence of events. The assumption can be easily enforced by inserting a dummy event command in the beginning of each function definition. We shall abbreviate a function type $T_1 \to T_2 \to \cdots \to T_n \to \texttt{int}$ as $\widetilde{T} \to \texttt{int}$. Note that, for simplicity, we omit non-deterministic choice $*$ and algebraic data structures such as lists, which are used in our running examples, from the language $\mathcal{L}$. It is easy to extend our type system in Sec. 4 with these features (see, e.g., [24, 26]).

The operational semantics of $\mathcal{L}$ is defined by the set of inductive and coinductive rules for deriving judgments of the form $e \Downarrow v \,\&\, \varpi$ and $e \Uparrow \bot \,\&\, \pi$. The former is for terminating evaluations and means that the evaluation of e terminates with the final result value v producing the finite sequence of events ϖ. The latter is for non-terminating evaluations and means that the evaluation of e diverges producing the infinite sequence of events π. The rules are analogous to the ones from [12] and deferred to the extended report [17].

4 Type System

4.1 First-Order Fixpoint Logic

The types in our dependent-refinement type system embed predicates in the first-order fixpoint logic over integers and finite and infinite event sequences. Fig. 4 shows the syntax. Meta-variable X represents *predicate variables*. $A(\widetilde{t})$ represents atomic formulas such as the equality on integers and sequences. f represents constants such as integers n, the empty sequence ϵ, and singleton sequences $\underline{\mathbf{a}}$ as well as functions such as the sequence concatenation and integer arithmetic operators. We write $\top$ and $\bot$ respectively for tautology and contradiction. The formula $\mu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi$ (resp. $\nu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi$) represents the least (resp. greatest) fixpoint (of the function $\lambda X.\lambda\widetilde{x} \in \widetilde{\mathcal{S}}.\phi$). We assume that X in $\mu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi$ and $\nu X(\widetilde{x}\colon \widetilde{\mathcal{S}}).\phi$ occurs only positively in ϕ. We sometimes omit sorts when they are obvious from the context. We define $(\lambda\widetilde{x}.\phi)(\widetilde{t}) \triangleq [\widetilde{t}/\widetilde{x}]\phi$ and write $p_1 \sqsubseteq p_2$ if $\forall\widetilde{x}.p_1(\widetilde{x}) \Rightarrow p_2(\widetilde{x})$ holds. We also write $\neg\lambda\widetilde{x}.\phi$ for $\lambda\widetilde{x}.\neg\phi$. We write $\models \phi$ if ϕ is valid. We define the formal semantics of this first-order fixpoint formulas in [17]. We write $fv(\phi)$ (resp. $fpv(\phi)$) for the set of free term (resp. predicate) variables in ϕ. We also define $fv(\lambda x.\phi) \triangleq fv(\phi) \setminus \{x\}$ and $fpv(\lambda x.\phi) \triangleq fpv(\phi)$.

$$
\begin{array}{rl}
\text{(effects) } \Phi & ::= (\lambda x \in \Sigma^*.\phi_\mu, \lambda x \in \Sigma^\omega.\phi_\nu) \\
\text{(effect qualified types) } \sigma & ::= (\tau \,\&\, \Phi) \\
\text{(dependent refinement types) } \tau & ::= \{x \mid \phi\} \mid (x : \tau) \to \sigma \\
\text{(type environments) } \Gamma & ::= \emptyset \mid \Gamma, x : \tau
\end{array}
$$

Figure 5. Syntax of types and effects

4.2 Syntax of Types and Effects

The syntax of types and effects is shown in Fig. 5. An *effect* Φ is a pair of predicates $\lambda x \in \Sigma^*.\phi_\mu$ and $\lambda x \in \Sigma^\omega.\phi_\nu$, which may contain free term and predicate variables. We write Φ^μ (resp. Φ^ν) for $\lambda x \in \Sigma^*.\phi_\mu$ (resp. $\lambda x \in \Sigma^\omega.\phi_\nu$). Φ^μ specifies a set of valid *finite* event sequences for terminating runs. On the other hand, Φ^ν specifies a set of valid *infinite* event sequences for non-terminating runs. We define the concatenation $\Phi_1 \cdot \Phi_2$ of effects Φ_1 and Φ_2 as follows.[1]

$$
\begin{array}{l}
(\lambda x \in \Sigma^*.\exists x_1, x_2 \in \Sigma^*.x = x_1 \cdot x_2 \wedge \Phi_1^\mu(x_1) \wedge \Phi_2^\mu(x_2), \\
\lambda x \in \Sigma^\omega.\Phi_1^\nu(x) \vee (\exists y \in \Sigma^*, z \in \Sigma^\omega.x = y \cdot z \wedge \Phi_1^\mu(y) \wedge \Phi_2^\nu(z)))
\end{array}
$$

We also define a special effect $\Phi_{val} \triangleq (\lambda x \in \Sigma^*.x = \epsilon, \lambda x \in \Sigma^\omega.\bot)$. Note that Φ_{val} is an identity with respect to $\cdot$, that is, $\Phi_{val} \cdot \Phi = \Phi \cdot \Phi_{val} = \Phi$ for any Φ. We often abbreviate $(\tau \,\&\, \Phi_{val})$ as τ.

An *effect qualified type* is of the form $(\tau \,\&\, \Phi)$ where τ is a dependent refinement type (described below) and Φ is an effect. Roughly, the qualified type $(\tau \,\&\, \Phi)$ is the type of expressions e such that, 1.) for all terminating run $e \Downarrow v \,\&\, \varpi$, v conforms to the type τ and ϖ satisfies Φ^μ, and 2.) for all non-terminating run $e \Uparrow \bot \,\&\, \pi$, ϖ satisfies Φ^ν. Sec. 4.3 formally defines the semantics of the qualified types.[2]

An *integer refinement type* $\{x \mid \phi\}$ is the type of integers x that satisfy the formula ϕ. We often abbreviate $\{x \mid \top\}$ as $\texttt{int}$. A *dependent function type* $(x : \tau) \to \sigma$ is the type of functions that take an argument x of the type τ and behave according to the return type σ. Note that the scope of x is within σ, and hence effects in σ can depend on the argument x. For example, $((x : \texttt{int}) \to (\texttt{int} \,\&\, \Phi_1) \,\&\, \Phi_2)$ is the type of expressions that exhibit event sequences conforming to Φ_2 when evaluated, and cause event sequences conforming to Φ_1 when applied to some integer argument x. We write $(\widetilde{x}\colon \widetilde{\tau}) \to (\tau \& \Phi)$ for $(x_1 : \tau_1) \to ((x_2 : \tau_2) \to (\cdots (x_n : \tau_n) \to (\tau \& \Phi) \cdots \& \Phi_{val}) \& \Phi_{val})$, that is, the latent effect of partially applying the function is Φ_{val}. Note that a type of such a form can be given to a recursive function $\texttt{rec}(f, \widetilde{x}, e)$ where $|\widetilde{x}| = |\widetilde{\tau}|$ as partial applications to f (i.e., applying less than $|\widetilde{x}|$ many arguments) do not raise any events. We abbreviate $(x : \tau) \to \sigma$ as $\tau \to \sigma$ when x does not occur in σ.

A type environment Γ is a sequence of variable bindings $x : \tau$. We define $\Gamma(x) \triangleq \tau$ if $x : \tau \in \Gamma$. We abbreviate $\Gamma, \nu : \{\nu \mid \phi\}$ as Γ, ϕ if $\nu \notin fv(\phi)$ and ν never occurs elsewhere. Note that type bindings in type environments and arguments of function types are of the form $(x : \tau)$ instead of $(x : \sigma)$. This is because the target language $\mathcal{L}$ is call-by-value, and hence variables are always bound to values whose evaluation never exhibits temporal effects.

We define auxiliary functions $sty(\sigma)$, $fv(\sigma)$, and $fpv(\sigma)$. $sty(\sigma)$ represents the simple type corresponding to the qualified type σ. $fv(\sigma)$ (resp. $fpv(\sigma)$) represents the set of free term (resp. predicate)

[1]Note that this generalizes the concatenation of non-dependent temporal effects from previous works [4, 12].

[2]Readers familiar with type and effect systems may find the qualified type notation atypical. We use the notation to simplify the presentation: for example, subtyping and subeffecting can be defined at once.

$$\llbracket \Gamma \vdash \sigma \rrbracket \triangleq \{e \mid \forall \theta \in sty(\Gamma).(\theta \models \Gamma) \Rightarrow \theta(e) \in \llbracket \theta(\sigma) \rrbracket\}$$
$$\llbracket (\tau \,\&\, \Phi) \rrbracket \triangleq \left\{ e \in sty(\tau) \;\middle|\; \begin{array}{l} (\forall \varpi, w.(e \Downarrow w \,\&\, \varpi) \Rightarrow \\ \quad (w \in \llbracket \tau \rrbracket) \wedge (\models \Phi^{\mu}(\varpi))) \wedge \\ \quad (\forall \pi.(e \Uparrow \bot \,\&\, \pi) \Rightarrow (\models \Phi^{\nu}(\pi))) \end{array} \right\}$$
$$\llbracket \{x \mid \phi\} \rrbracket \triangleq \{n \mid \models [n/x]\phi\}$$
$$\llbracket (x:\tau) \to \sigma \rrbracket \triangleq \{w \in sty(\tau \to \sigma) \mid \forall w' \in \llbracket \tau \rrbracket . w\, w' \in \llbracket [w'/x]\sigma \rrbracket\}$$

Figure 6. Semantic typing

variables that occur in σ. The definitions are standard and deferred to the extended report [17]. We extend the notions to type environments and define $sty(\Gamma)$, $fv(\Gamma)$, and $fpv(\Gamma)$ in the obvious way.

We remark that our type and effects are essentially the extension of the types from the previous work on dependent-refinement type systems [12, 20, 23, 24, 28, 30] with dependent temporal effects which are first-order fixpoint logic predicates on program values and (finite and infinite) event sequences. Note that, as in the previous work, the dependent types are restricted to facilitate (semi-)automated reasoning via modern SMT and constraint solving techniques. Namely, the types can only depend on non-function and effect-free terms.

4.3 Semantic Typing

To formalize the type soundness theorem (cf. Theorem 4.1), we define the semantics of qualified types. Fig. 6 defines the semantics. Here, w is a meta-variable ranging over closed values (i.e., $fv(w) = \emptyset$). We write $e \in T$ if the expression e has the simple type T. Similarly, we write $w \in T$ if the closed value w has the type T. We write $\theta \in E$ for a simple type environment E and a closed value substitution θ if $\mathrm{dom}(\theta) = \mathrm{dom}(E)$ and $\theta(x) \in E(x)$ for any $x \in \mathrm{dom}(E)$. Also, we write $\theta \models \Gamma$ if $\mathrm{dom}(\theta) = \mathrm{dom}(\Gamma)$ and $\forall (x:\tau) \in \Gamma.\theta(x) \in \llbracket \theta(\tau) \rrbracket$ hold.

Note that $\llbracket \Gamma \vdash \sigma \rrbracket$ denotes the set of open expressions that behave according to σ under an environment conforming to Γ. Similarly, $\llbracket \sigma \rrbracket$ (resp. $\llbracket \tau \rrbracket$) denotes the set of closed expressions (resp. values) that behave according to σ (resp. τ). For instance, for $\tau = (x : \{u \mid u \geq 0\}) \to (\{v \mid v > x\} \,\&\, \Phi)$ where $\Phi = (\lambda z \in \Sigma^*.z \in \underline{\mathbf{a}}^x, \lambda z \in \Sigma^{\omega}.\bot)$, $\llbracket \tau \rrbracket$ are the set of closed functions from integers to integers which, when given a non-negative integer x as the argument, raises the event $\underline{\mathbf{a}}$ x many times and returns an integer greater than x.

We also define the semantics of subtyping relation as follows, which says when a (qualified) type is a subtype of another in the given type environment.

$$\llbracket \Gamma \vdash \sigma_1 <: \sigma_2 \rrbracket \triangleq \forall \theta \in sty(\Gamma).(\theta \models \Gamma) \Rightarrow \llbracket \theta(\sigma_1) \rrbracket \subseteq \llbracket \theta(\sigma_2) \rrbracket$$
$$\llbracket \Gamma \vdash \tau_1 <: \tau_2 \rrbracket \triangleq \forall \theta \in sty(\Gamma).(\theta \models \Gamma) \Rightarrow \llbracket \theta(\tau_1) \rrbracket \subseteq \llbracket \theta(\tau_2) \rrbracket$$

Sec. 4.4 shows the rules for deriving subtyping judgments.

4.4 Typing Rules

Fig. 7 shows the typing rules. The rules derive judgments of the form $\Gamma \vdash e\colon \sigma$, saying that e behaves according to σ under a value environment conforming to Γ.

We describe the typing rules. The rules T-Const for typing integer constants, T-VInt for typing integer-type term variables, and T-VFun for typing function-type variables, T-Op for typing integer

$$\frac{}{\Gamma \vdash n\colon (\{x \mid x = n\} \,\&\, \Phi_{val})}\ \textsc{T-Const}$$

$$\frac{sty(\Gamma(x)) = \mathtt{int}}{\Gamma \vdash x\colon (\{u \mid u = x\} \,\&\, \Phi_{val})}\ \textsc{T-VInt} \qquad \frac{sty(\Gamma(x)) \neq \mathtt{int}}{\Gamma \vdash x\colon (\Gamma(x) \,\&\, \Phi_{val})}\ \textsc{T-VFun}$$

$$\frac{x \notin fv(\tau_2) \cup fv(\Phi_2) \quad \Gamma \vdash e_1\colon (\tau_1 \,\&\, \Phi_1) \quad \Gamma, x:\tau_1 \vdash e_2\colon (\tau_2 \,\&\, \Phi_2)}{\Gamma \vdash \mathtt{let}\ x\mathtt{=}\ e_1\ \mathtt{in}\ e_2\colon (\tau_2 \,\&\, \Phi_1 \cdot \Phi_2)}\ \textsc{T-Let}$$

$$\frac{\Gamma \vdash v_1\colon ((x:\tau) \to (\tau' \,\&\, \Phi) \,\&\, \Phi_{val}) \quad \Gamma \vdash v_2\colon (\tau \,\&\, \Phi_{val})}{\Gamma \vdash v_1\ v_2\colon [v_2/x](\tau' \,\&\, \Phi)}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash v_1\colon (\mathtt{int} \,\&\, \Phi_{val}) \quad \Gamma \vdash v_2\colon (\mathtt{int} \,\&\, \Phi_{val})}{\Gamma \vdash v_1\ op\ v_2\colon (\{x \mid x = v_1\ op\ v_2\} \,\&\, \Phi_{val})}\ \textsc{T-Op}$$

$$\frac{\Gamma, v = 0 \vdash e_1\colon \sigma \quad \Gamma, v \neq 0 \vdash e_2\colon \sigma}{\Gamma \vdash \mathtt{ifz}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\colon \sigma}\ \textsc{T-If}$$

$$\frac{\Phi = (\lambda x \in \Sigma^*.x = \underline{\mathbf{a}}, \lambda x \in \Sigma^{\omega}.\bot)}{\Gamma \vdash \mathtt{ev}[\underline{\mathbf{a}}]\colon (\{x \mid x = 0\} \,\&\, \Phi)}\ \textsc{T-Event}$$

$$\frac{\begin{array}{c} \tau_f' = (\widetilde{x}\colon \widetilde{\tau}) \to (\tau \,\&\, (\lambda x \in \Sigma^*.X_{\mu}(\widetilde{x}, x), \lambda x \in \Sigma^{\omega}.X_{\nu}(\widetilde{x}, x))) \\ \Gamma, f:\tau_f', \widetilde{x}:\widetilde{\tau} \vdash e\colon (\tau \,\&\, \Phi) \\ q_{\mu} = \mu X_{\mu}(\widetilde{x}, x).\Phi^{\mu}(x) \quad q_{\nu} = \nu X_{\nu}(\widetilde{x}, x).[q_{\mu}/X_{\mu}]\Phi^{\nu}(x) \\ \tau_f = (\widetilde{x}\colon \widetilde{\tau}) \to (\tau \,\&\, (\lambda x \in \Sigma^*.q_{\mu}(\widetilde{x}, x), \lambda x \in \Sigma^{\omega}.q_{\nu}(\widetilde{x}, x))) \end{array}}{\Gamma \vdash \mathtt{rec}(f, \widetilde{x}, e)\colon (\tau_f \,\&\, \Phi_{val})}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash e\colon \sigma_1 \quad \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash e\colon \sigma_2}\ \textsc{T-Sub} \qquad \frac{\Vdash \lfloor \Gamma \vdash \phi_1 \Rightarrow \phi_2 \rfloor}{\Gamma \vdash \{u \mid \phi_1\} <: \{u \mid \phi_2\}}\ \textsc{S-Int}$$

$$\frac{\begin{array}{c} \Vdash \lfloor \Gamma \vdash \forall x \in \Sigma^*.\Phi_1^{\mu}(x) \Rightarrow \Phi_2^{\mu}(x) \rfloor \\ \Gamma \vdash \tau_1 <: \tau_2 \qquad \Vdash \lfloor \Gamma \vdash \forall x \in \Sigma^{\omega}.\Phi_1^{\nu}(x) \Rightarrow \Phi_2^{\nu}(x) \rfloor \end{array}}{\Gamma \vdash (\tau_1 \,\&\, \Phi_1) <: (\tau_2 \,\&\, \Phi_2)}\ \textsc{S-Qual}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma, x:\tau_2 \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash (x:\tau_1) \to \sigma_1 <: (x:\tau_2) \to \sigma_2}\ \textsc{S-Fun}$$

Figure 7. Typing and subtyping rules

operations, T-If for typing conditional branches are straightforward extension of those from the previous work on dependent-refinement type systems [12, 20, 23, 24]. Note that Φ_{val} is assigned as the effect of the expression in T-Const, T-VInt, T-VFun, and T-Op because these expressions always terminate and raise no events.

T-Fun types recursive function definitions. As described in Sec. 2, the main idea here is to introduce predicate variables X_{μ} and X_{ν} respectively representing the finite and infinite effects of the recursive function f being typed. Then, the rule types the body of the function, e, under the environment that f has these temporal effects (expressed by the binding $f : \tau_f'$), to obtain the qualified type $(\tau \,\&\, (\Phi^{\mu}, \Phi^{\nu}))$. Then, the latent effect of the function can be obtained by, for the finite part, taking the least fixpoint of Φ^{μ} (i.e., $q_{\mu} = \mu X_{\mu}(\widetilde{x}, x).\Phi^{\mu}(x)$), and the greatest fixpoint for the infinite part using the least fixpoint solution for X_{μ} appearing in the formula (i.e., $\nu X_{\nu}(\widetilde{x}, x).[q_{\mu}/X_{\mu}]\Phi^{\nu}(x)$). The rule adopts the one from [4] and extend it to dependent type and effects.

The rule T-Let for typing let expressions extends a similar rule from the previous work [4, 12]. Note that concatenation is used to obtain the effect of the expression in the conclusion, correctly accounting for the fact that the possible event traces of the expression are the concatenation of those of e_1 and e_2. T-App types function applications, and is a straightforward extension of those from the previous work on dependent-refinement type systems and (non-dependent) type-and-effect systems. As in a standard type-and-effect system, the latent effect of the function, Φ, becomes the effect of the function application. Note here that the variable x may occur in the latent effect, which is substituted by v_2 in the conclusion to account for the dependency. T-Event types event raising operations and is self-explanatory. Finally, T-Sub is the subsumption rule.

Fig. 7 also shows the rules for deriving subtyping judgments. There, auxiliary functions $\lfloor\Gamma\rfloor$ and $\lfloor\Gamma \vdash \phi\rfloor$ are defined by:

$$\lfloor\emptyset\rfloor \triangleq \top \qquad \lfloor\Gamma, x : \{y \mid \phi\}\rfloor \triangleq \lfloor\Gamma\rfloor \wedge [x/y]\phi$$
$$\lfloor\Gamma \vdash \phi\rfloor \triangleq \lfloor\Gamma\rfloor \Rightarrow \phi \qquad \lfloor\Gamma, x : (y : \tau) \to \sigma\rfloor \triangleq \lfloor\Gamma\rfloor$$

The rules S-Int and S-Fun for subtyping refinement integer types and dependent function types are equivalent to those from the previous work on dependent-refinement type systems. The rule S-Qual is for subtyping effect qualified types. It asserts that the type part of the qualified types τ_1 and τ_2 are in the subtyping relationship. Further, it checks that the left effect Φ_1 is a *subeffect* of the right effect Φ_2. The subeffecting relation checks that the finite (resp. infinite) part of Φ_1 logically implies the finite (resp. infinite) part of Φ_2, under the assertions implied by the type environment Γ. For example, in the typing of `messenger` from Sec. 2, a subtyping judgment $\Gamma \vdash (\tau \,\&\, \Phi_{\texttt{send\_msgs}})$ <: $(\texttt{int} \,\&\, \Phi'_{\texttt{send\_msgs}})$ is discharged where $\Gamma = n : \{x \mid x \geq 0\}$, τ is the refinement integer type obtained for `send_msgs` n, and $\Phi_{\texttt{send\_msgs}}$ and $\Phi'_{\texttt{send\_msgs}}$ are from Fig. 1. S-Qual checks the subtyping by asserting the validity of $n \geq 0 \Rightarrow \forall x \in \Sigma^*.\Phi^{\mu}_{\texttt{send\_msgs}}(x) \Rightarrow \Phi'^{\mu}_{\texttt{send\_msgs}}(x)$ and $n \geq 0 \Rightarrow \forall x \in \Sigma^*.\Phi^{\nu}_{\texttt{send\_msgs}}(x) \Rightarrow \Phi'^{\nu}_{\texttt{send\_msgs}}(x)$. Sec. 5 shows the deductive system for solving such predicate fixpoint logic constraints.

We show that the type system is sound, that is, the judgments derived by the typing rules respect the semantics. We define *predicate substitution* ρ to be a finite map from predicate variables to closed predicates.

Theorem 4.1. *If* $\Gamma \vdash e : \sigma$*, then* $e \in [\![\rho(\Gamma) \vdash \rho(\sigma)]\!]$ *for any predicate substitution* ρ *with* $\mathrm{dom}(\rho) = \mathit{fpv}(\Gamma) \cup \mathit{fpv}(\sigma)$.

We remark that the soundness holds for any background first-order theory supporting basic integer arithmetic (i.e., those in $\mathcal{L}$) and concatenations of finite and infinite string over a finite alphabet. Hence, our system can reap the benefits of recent advances in automated deduction for various theories on integers, finite and infinite string, and combinations thereof [1, 5].

5 Deductive Proof System For First-Order Fixpoint Logic

We now present our deductive system for the first-order fixpoint logic introduced in Sec. 4.1. The deductive system is intended, but not limited, to be used to discharge proof obligations that arise during the process of type checking and inference for the type system presented in Sec. 4.4.

The deductive system comprises rules for reasoning via invariants and well-founded relations, and is able to solve formulas containing both least and greatest fixpoints. The key idea is to soundly approximate formulas with fixpoints as formulas without fixpoints,

$$\frac{\models \psi}{\Vdash \psi}\ \textsc{Fp-Valid} \qquad \frac{\models [(\lambda\widetilde{x}.\psi')/X]\psi \Rightarrow \psi' \quad \Vdash C^-[[\widetilde{t}/\widetilde{x}]\psi']}{\Vdash C^-[(\mu X(\widetilde{x}).\psi)(\widetilde{t})]}\ \textsc{Fp-Lfp}^-$$

$$\frac{\models \psi' \Rightarrow [(\lambda\widetilde{x}.\psi')/X]\psi \quad \Vdash C^+[[\widetilde{t}/\widetilde{x}]\psi']}{\Vdash C^+[(\nu X(\widetilde{x}).\psi)(\widetilde{t})]}\ \textsc{Fp-Gfp}^+$$

$$\frac{X(\widetilde{x}); p_1; p_2; \top \downarrow \mathit{nnf}(\psi) \quad \Vdash C^+[p_1(\widetilde{t})] \quad \models \mathit{WF}(p_2)}{\Vdash C^+[(\mu X(\widetilde{x}).\psi)(\widetilde{t})]}\ \textsc{Fp-Lfp}^+$$

$$\frac{X(\widetilde{x}); p_1; p_2; \top \,\bar{\uparrow}\, \mathit{nnf}(\psi) \quad \Vdash C^-[\neg p_1(\widetilde{t})] \quad \models \mathit{WF}(p_2)}{\Vdash C^-[(\nu X(\widetilde{x}).\psi)(\widetilde{t})]}\ \textsc{Fp-Gfp}^-$$

Figure 8. Proof rules for the fixpoint logic

which may be checked by off-the-shelf first-order theorem provers (SMT solvers) supporting the theories of integers and finite and infinite strings over finite alphabet [1, 5].

A judgment $\Vdash \phi$ of the deductive system means that ϕ is valid. The derivation rules for $\Vdash \phi$ are shown in Fig. 8. There, meta-variable ψ ranges over formulas not containing fixpoint formulas (i.e., those of the form $(\mu X(\widetilde{x}).\phi)(\widetilde{t})$ and $(\nu X(\widetilde{x}).\phi)(\widetilde{t})$). The formula $\mathit{nnf}(\psi)$ is the negation normal form of ψ, and $\models \mathit{WF}(p)$ means that the predicate $p = \lambda\widetilde{x}.\phi$ is *well-founded*, that is, the arity of p is $2 \times n$ for some n and there is no infinite sequence $\widetilde{t}_1, \widetilde{t}_2, \ldots$ such that $|\widetilde{t}_i| = n$ and $p(\widetilde{t}_i, \widetilde{t}_{i+1})$ holds for all $i \geq 1$. C^+ (resp. C^-) is a formula context whose hole occurs in a positive (resp. negative) position.

We now describe the rules. The rule Fp-Valid checks the validity directly, and is applied when the given formula does not contain fixpoint formulas. $\textsc{Fp-Lfp}^-$ over-approximates a least fixpoint $\mu X(\widetilde{x}).\psi$ that occurs in a negative position with a pre-fixpoint $\lambda\widetilde{x}.\psi'$ of the function $\lambda X.\lambda\widetilde{x}.\psi$. Note here that ψ and ψ' do not contain fixpoints. An example of $\textsc{Fp-Lfp}^-$ can be seen in Sec. 2, where we discuss proof subtree **A**. Meanwhile, $\textsc{Fp-Gfp}^+$ is a dual rule and it under-approximates a greatest fixpoint $\nu X(\widetilde{x}).\psi$ that occurs in a positive position with a post-fixpoint $\lambda\widetilde{x}.\psi'$ of $\lambda X.\lambda\widetilde{x}.\psi$.

By contrast, $\textsc{Fp-Lfp}^+$ under-approximates a least fixpoint that occurs in a positive position with a predicate p_1. Here, the auxiliary judgment $X(\widetilde{x}); p_1; p_2; \psi' \downarrow \psi$ is used to check that the well-founded predicate p_2 witnesses that p_1 under-approximates the least fixpoint $\mu X(\widetilde{x}).\psi' \Rightarrow \psi$. In a dual manner, $\textsc{Fp-Gfp}^-$ over-approximates a greatest fixpoint that occurs in a negative position with a predicate $\neg p_1$. The auxiliary judgment $X(\widetilde{x}); p_1; p_2; \psi' \,\bar{\uparrow}\, \psi$ checks that the well-founded predicate p_2 witnesses that $\neg p_1$ over-approximates the greatest fixpoint $\nu X(\widetilde{x}).\psi' \wedge \psi$. An example of $\textsc{Fp-Gfp}^-$ can be seen in Sec. 2, where we discuss proof subtree **B**.

The rules in Fig. 8 reduce fixpoints to predicates but, in two cases, lead to side conditions that the predicates indeed approximate the fixpoints. These conditions are treated themselves as judgments in the auxiliary relations $X(\widetilde{x}); p_1; p_2; \psi' \downarrow \psi$ and $X(\widetilde{x}); p_1; p_2; \psi' \,\bar{\uparrow}\, \psi$, defined in Fig. 9. There, we maintain the invariants that ψ is in the negation normal form, ψ' does not contain X, and X may occur only positively in ψ. The rules let us then manipulate the judgments to further reduce to predicate reasoning. The rules $\textsc{Apx}^{\mu}$-$\wedge$ and $\textsc{Apx}^{\mu}$-$\forall$ are similar to standard ones for first-order logic. $\textsc{Apx}^{\mu}$-$\vee$ splits a judgment with the succedent of the form $\psi_1 \vee \psi_2$ into two judgments: one with the succedent ψ_1 and the antecedent conjuncted with ψ'_1 and the other with the succedent

$$\frac{\models p_1(\widetilde{x}) \wedge \psi' \Rightarrow \psi}{X(\widetilde{x}); p_1; p_2; \psi' \downarrow \psi}\ \textsc{Apx}^{\mu}\text{-}\textsc{Base}$$

$$\frac{\models p_1(\widetilde{x}) \wedge \psi \Rightarrow p_1(\widetilde{t}) \wedge p_2(\widetilde{x}, \widetilde{t})}{X(\widetilde{x}); p_1; p_2; \psi \downarrow X(\widetilde{t})}\ \textsc{Apx}^{\mu}\text{-}\textsc{Rec}$$

$$\frac{X(\widetilde{x}); p_1; p_2; \psi \downarrow \psi_1 \quad X(\widetilde{x}); p_1; p_2; \psi \downarrow \psi_2}{X(\widetilde{x}); p_1; p_2; \psi \downarrow \psi_1 \wedge \psi_2}\ \textsc{Apx}^{\mu}\text{-}\wedge$$

$$\frac{\begin{array}{c}\models (p_1(\widetilde{x}) \wedge \psi) \Rightarrow (\psi_1' \vee \psi_2') \quad fv(\psi_i') \subseteq \{\widetilde{x}\} \quad X \notin fpv(\psi_i') \\ X(\widetilde{x}); p_1; p_2; \psi \wedge \psi_i' \downarrow \psi_i \quad (i = 1, 2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi \downarrow \psi_1 \vee \psi_2}\ \textsc{Apx}^{\mu}\text{-}\vee$$

$$\frac{\begin{array}{c}X(\widetilde{x}); p_1; p_2; \psi' \downarrow [x'/x]\psi \\ x' \notin fv(\psi') \cup fv(\psi) \cup \{\widetilde{x}\} \cup fv(p_1) \cup fv(p_2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi' \downarrow \forall x.\psi}\ \textsc{Apx}^{\mu}\text{-}\forall$$

$$\frac{\begin{array}{c}\models (p_1(\widetilde{x}) \wedge \psi') \Rightarrow \exists x'.\psi'' \quad fv(\psi'') \subseteq \{\widetilde{x}, x'\} \quad X \notin fpv(\psi'') \\ X(\widetilde{x}); p_1; p_2; \psi' \wedge \psi'' \downarrow [x'/x]\psi \\ x' \notin fv(\psi') \cup fv(\psi) \cup \{\widetilde{x}\} \cup fv(p_1) \cup fv(p_2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi' \downarrow \exists x.\psi}\ \textsc{Apx}^{\mu}\text{-}\exists$$

$$\frac{\models p_1(\widetilde{x}) \wedge \psi' \Rightarrow \neg\psi}{X(\widetilde{x}); p_1; p_2; \psi' \mathrel{\bar{\uparrow}} \psi}\ \textsc{Apx}^{\nu}\text{-}\textsc{Base}$$

$$\frac{\models p_1(\widetilde{x}) \wedge \psi \Rightarrow p_1(\widetilde{t}) \wedge p_2(\widetilde{x}, \widetilde{t})}{X(\widetilde{x}); p_1; p_2; \psi \mathrel{\bar{\uparrow}} X(\widetilde{t})}\ \textsc{Apx}^{\nu}\text{-}\textsc{Rec}$$

$$\frac{\begin{array}{c}\models (p_1(\widetilde{x}) \wedge \psi) \Rightarrow (\psi_1' \vee \psi_2') \quad fv(\psi_i') \subseteq \{\widetilde{x}\} \quad X \notin fpv(\psi_i') \\ X(\widetilde{x}); p_1; p_2; \psi \wedge \psi_i' \mathrel{\bar{\uparrow}} \psi_i \quad (i = 1, 2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi \mathrel{\bar{\uparrow}} \psi_1 \wedge \psi_2}\ \textsc{Apx}^{\nu}\text{-}\wedge$$

$$\frac{X(\widetilde{x}); p_1; p_2; \psi \mathrel{\bar{\uparrow}} \psi_1 \quad X(\widetilde{x}); p_1; p_2; \psi \mathrel{\bar{\uparrow}} \psi_2}{X(\widetilde{x}); p_1; p_2; \psi \mathrel{\bar{\uparrow}} \psi_1 \vee \psi_2}\ \textsc{Apx}^{\nu}\text{-}\vee$$

$$\frac{\begin{array}{c}\models (p_1(\widetilde{x}) \wedge \psi') \Rightarrow \exists x'.\psi'' \quad fv(\psi'') \subseteq \{\widetilde{x}, x'\} \quad X \notin fpv(\psi'') \\ X(\widetilde{x}); p_1; p_2; \psi' \wedge \psi'' \mathrel{\bar{\uparrow}} [x'/x]\psi \\ x' \notin fv(\psi') \cup fv(\psi) \cup \{\widetilde{x}\} \cup fv(p_1) \cup fv(p_2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi' \mathrel{\bar{\uparrow}} \forall x.\psi}\ \textsc{Apx}^{\nu}\text{-}\forall$$

$$\frac{\begin{array}{c}X(\widetilde{x}); p_1; p_2; \psi' \mathrel{\bar{\uparrow}} [x'/x]\psi \\ x' \notin fv(\psi') \cup fv(\psi) \cup \{\widetilde{x}\} \cup fv(p_1) \cup fv(p_2)\end{array}}{X(\widetilde{x}); p_1; p_2; \psi' \mathrel{\bar{\uparrow}} \exists x.\psi}\ \textsc{Apx}^{\nu}\text{-}\exists$$

Figure 9. Fixpoint approximation rules

ψ_2 and the antecedent conjuncted with ψ_2' for some ψ_1' and ψ_2' such that $\psi_1' \vee \psi_2'$ holds provided the original antecedent does. Apx$^{\mu}$-$\exists$ generalizes Apx$^{\mu}$-$\vee$ to judgments with the succedent of the form $\exists x.\psi$. Apx$^{\mu}$-Base exploits an external validity checker for formulas without fixpoints to discharge a judgment with the succedent ψ, by assuming the predicate variables (including X) that occur in ψ uninterpreted. Apx$^{\mu}$-Rec checks that the arguments $\widetilde{t}$ to the recursive occurrence of X satisfy p_1 and the pair $(\widetilde{x}, \widetilde{t})$ satisfies the well-founded predicate p_2 for any sequence of arguments $\widetilde{x}$ of X, in order to ensure that X is interpreted as an unbounded but finite unfolding of $\mu X(\widetilde{x}).\psi$. The rules Apx$^{\nu}$-$*$ are defined in a dual manner to Apx$^{\mu}$-$*$. Note that the roles of $\wedge$ and $\vee$ (also $\forall$ and $\exists$) are switched. In the `send_msgs` derivation in Fig. 1, we used the greatest fixpoint rules for conjunction (Apx$^{\nu}$-$\wedge$), existential quantification (Apx$^{\nu}$-$\exists$), and conjunction again.

Lemma 5.1 shows that the approximation rules correctly under/over approximate least/greatest fixpoints. The soundness result (Theorem 5.2 immediately follows.

Lemma 5.1. *Suppose that ψ is in negation normal form, $X \notin fpv(\psi')$, and $\models WF(p_2)$. We have:*

1. *if $X(\widetilde{x}); p_1; p_2; \psi' \downarrow \psi$, then $\models p_1(\widetilde{x}) \Rightarrow (\mu X(\widetilde{x}).\neg\psi' \vee \psi)(\widetilde{x})$,*
2. *if $X(\widetilde{x}); p_1; p_2; \psi' \mathrel{\bar{\uparrow}} \psi$, then $\models (\nu X(\widetilde{x}).\psi' \wedge \psi)(\widetilde{x}) \Rightarrow \neg p_1(\widetilde{x})$.*

Theorem 5.2. *If $\Vdash \phi$, then $\models \phi$.*

The decidability of the deduction problem depends on the background first-order theory. It is undecidable for the fragment used by our type system (indeed, it is already so with just linear integer arithmetic). See [17] for details.

6 Related Work

Verification of higher-order programs is an active topic of research. In recent years, numerous approaches have been proposed for automatically (or semi-automatically) verifying a wide range of temporal properties, including safety properties [6, 9, 19, 20, 23–25, 31, 32], termination [13, 28], non-termination [3, 14], and properties expressed in linear μ-calculus [12, 16].

However, the existing proposals employ rather disparate techniques to verify the different classes of properties. For instance, the safety property verification method of [9] applies predicate abstraction with CEGAR to iteratively reduce the problem to that of higher-order model checking [7, 18], whereas the termination verification method of [13] and linear μ-calculus verification method of [16] are based on a reduction to binary reachability analysis via program transformation. By contrast, we propose a unified type-based approach to verify an expressive range of temporal properties given as dependent-refinement types carrying dependent temporal effects. The class of properties supported by our method subsumes those considered in the previous work mentioned above aside from the non-termination property handled by [3, 14]. (Non-termination is not within the scope of our work because it is a branching property. See below for further discussion.)

An important classes of properties that are not addressed in this paper are *branching* properties, such as those expressible in the branching μ-calculus. Sound and complete methods for the class exist for well-typed finite-data higher-order programs (i.e., higher-order recursion schemes) [8, 18]. For infinite-data higher-order programs, a recent work by Unno et al. [26] proposes a type system that can uniformly deduce some restricted forms of branching properties such as conditional non-safety and conditional non-termination. However, their work does not address general temporal properties (even for the linear subclass). We leave the extension to branching properties as future work.

The dependent effects of our work are inspired by temporal effects from the previous work on type-and-effect systems for temporal property verification [4, 12, 21]. Like in our work, temporal effects facilitate compositional reasoning whereby the temporal behavior of program sub-terms are summarized as effects and combined to derive those of larger parts. However, the effects there were non-dependent and also often coarsely over-approximate the actual temporal behavior. For instance, [4, 21] only allow (ω-)regular sets of event sequences, and [12] (without oracles) always assigns Σ^ω as the infinite effect part of a recursive function. Our work extends the effects to dependent effects, which are fixpoint predicates on event sequences and program values that can precisely capture the temporal behavior, thereby enabling precise specification and verification of rich value-dependent temporal properties.

Our dependent temporal effects are first-order *predicate* fixpoint logic formulas on event sequences and program values. While fixpoint logics such as μ-calculus are prevalent in temporal property verification, most existing works only focus on the propositional fragment (even for verification of infinite state systems [12, 16]), and few considers temporal properties specified in a general predicate fixpoint logic. In [22], a system for deriving entailments in a predicate fixpoint logic using well-founded induction is presented. However, verification is not within the scope of their work.

An orthogonal direction of extension to the fixpoint logic is to include higher-order propositions (or predicates) [15, 29]. In a recent work, Kobayashi et al. [11] have proposed to apply such higher-order fixpoint logic (HFL) for verification of higher-order programs. Similar to our approach, they encode the verification problem as problems in the fixpoint logic. More concretely, their approach encodes the given higher-order program as a HFL formula so that the verification problem is reduced to a model checking problem for HFL. However, their work does not present concrete means to solve the obtained fixpoint logic problem (besides the case for the propositional fragment which they show to be equivalent to model checking of higher-order recursion schemes [10, 11]), whereas we propose a deductive system for solving the fixpoint logic constraints generated by the type-based verification process. On the other hand, compared to our work that uses first-order fixpoint logic, the use of higher-order logic may prove advantageous in being able to more naturally model verification problems for higher-order programs, analogous to the recent proposal of *higher-order constrained Horn clauses* for safety verification of higher-order programs [2]. We leave as future work for a deeper investigation of the relation.

7 Conclusion and Future Work

We have presented a novel method for reasoning about the temporal properties of higher-order programs. We use a type-based, compositional approach that is, in contrast to prior work [12], nonetheless amenable to algorithmic verification. Also, our treatment with effect predicates and predicate variables, has led to least/greatest-fixpoint typing rules that do not sacrifice precision, as was the case in other prior work. We also present a deductive fixpoint proof system that allows us to introduce approximations in the form of invariants and well-founded relations.

In future work, we plan to build on our type system and develop type inference algorithms, automating our type system and deductive fixpoint proof system. We also plan to explore relationships between our dependent temporal events and works on resource analysis, as discussed with the amortized complexity example at the end of Sec. 2.

Acknowledgments

We thank Naoki Kobayashi and anonymous referees for helpful comments and suggestions. This research was supported in part by MEXT Kakenhi 15H05706, 16H05856, 17H01720, and 17H01723; JSPS Core-to-Core Program, A.Advanced Research Networks; JSPS Bilateral Collaboration Research; the National Science Foundation (NSF) award #1618542; and the Office of Naval Research (ONR) award #N000141712787.

References

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV*, 2014.

[2] T. C. Burn, C. L. Ong, and S. J. Ramsay. Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL), 2018.

[3] K. Hashimoto and H. Unno. Refinement type inference via horn constraint optimization. In *SAS*, 2015.

[4] M. Hofmann and W. Chen. Abstract interpretation from Büchi automata. In *CSL-LICS*, 2014.

[5] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL), 2018.

[6] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *CAV*, 2011.

[7] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.

[8] N. Kobayashi and C. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, 2009.

[9] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, 2011.

[10] N. Kobayashi, É. Lozes, and F. Bruse. On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In *POPL*, 2017.

[11] N. Kobayashi, T. Tsukada, and K. Watanabe. Higher-order program verification via HFL model checking. In *ESOP*, 2018.

[12] E. Koskinen and T. Terauchi. Local temporal reasoning. In *CSL-LICS*, 2014.

[13] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *ESOP*, 2014.

[14] T. Kuwahara, R. Sato, H. Unno, and N. Kobayashi. Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In *CAV*, 2015.

[15] M. Lange, É. Lozes, and M. V. Guzmán. Model-checking process equivalences. *Theor. Comput. Sci.*, 560, 2014.

[16] A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. Temporal verification of higher-order functional programs. In *POPL*, 2016.

[17] Y. Nanjo, H. Unno, E. Koskinen, and T. Terauchi. A fixpoint logic and dependent effects for temporal property verification. Extended version, available from http://www.cs.tsukuba.ac.jp/~uhiro/.

[18] C. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.

[19] C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, 2011.

[20] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[21] C. Skalka, S. F. Smith, and D. V. Horn. Types and trace effects of higher order programs. *J. Funct. Program.*, 18(2), 2008.

[22] C. Sprenger and M. Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In *FOSSACS*, 2003.

[23] T. Terauchi. Dependent types from counterexamples. In *POPL*, 2010.

[24] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.

[25] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, 2013.

[26] H. Unno, Y. Satake, and T. Terauchi. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *PACMPL*, 2(POPL), 2018.

[27] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Logic*, 51(1-2), 1991.

[28] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *ICFP*, 2014.

[29] M. Viswanathan and R. Viswanathan. A higher order modal fixed point logic. In *CONCUR*, 2004.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

[31] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *VMCAI*, 2013.

[32] H. Zhu, A. V. Nori, and S. Jagannathan. Learning refinement types. In *ICFP*, 2015.