A Fixpoint Logic and Dependent Effects for Temporal Property Verification

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

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

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

Tachio Terauchi
Waseda University
terauchi@waseda.jp

Abstract

Existing approaches to temporal verification of higher-order functional programs have either sacrificed compositionalality 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:

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, 19, 22–24], termination [13, 27], non-termination [14], or finite data [17]. 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 [26] can be lifted to the higher-order setting [16]. Still other works have permitted reasoning about angelic-vs-demonic nondeterminism [25].

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, 20]. 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 \((\tau \& (\Phi^\mu, \Phi^\nu))\) where we use dependent-refinement types and, as in prior work [4, 12, 20], the effects are a pair: \(\Phi^\mu\) corresponding to the finite effects and \(\Phi^\nu\) 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 \(\Sigma^*\) and a predicate on \(\Sigma^\omega\) over some alphabet of events \(\Sigma\). 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 \(\text{foo}\) is given by the pair \((\Phi^\mu_{\text{foo}}, \Phi^\nu_{\text{foo}})\) defined as:

\[
\Phi^\mu_{\text{foo}} \triangleq \lambda x. \bot \quad \Phi^\nu_{\text{foo}} \triangleq \lambda x. x \in ((\text{Ready} \cdot \text{Send}^\infty) | \text{Wait})^\omega
\]
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 $\text{Ready}$ event and n $\text{Send}$ events or (ii) a single $\text{Wait}$ event. Notice, in particular, that n is a parameter to $\text{foo}$, making this effect dependent with respect to $\text{foo}$’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 $rec(f, x, e)$ with two constraints: a least fixpoint constraint relating finite effects $Φ^μ$ to the finite effects of $rec(f, x, e)$ and a greatest fixpoint constraint relating the infinite effects $Φ^ν$ to the infinite effects of $rec(f, x, e)$. These effects of recursive functions have the form:

$$Φ^μ_{\text{foo}} = \lambda x.(\mu X_n(n, x) \ldots) (n, x) \quad Φ^ν_{\text{foo}} = \lambda x.(\nu X_n(n, x) \ldots) (n, x)$$

where $X_n$ and $X_\nu$ 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, 20] and algorithmic verification approaches [8, 9, 13, 16, 19, 22–25]. 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)
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.
The second component $\Phi_{send msgs}'$ describes the infinite effects of $send msgs$. Not surprisingly, a greatest fixpoint equation is used, with predicate variable $X_v$ 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.

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

---

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Typing Rules and Final Effect Approximations</th>
</tr>
</thead>
<tbody>
<tr>
<td>let rec until_ready () = if * then (event[Ready]; ()) else (event[Wait]; until_ready ())</td>
<td></td>
</tr>
<tr>
<td>let rec send_msgs n = if n = 0 then () else (event[send]; sendmsgs (n-1))</td>
<td></td>
</tr>
</tbody>
</table>

### (a) Source Code

- `let rec until_ready () =` defines a recursive function `until_ready`.
- `if * then (event[Ready]; ()) else (event[Wait]; until_ready ())` is the body of the recursive function, where `*` is a placeholder for future updates.
- `let rec send_msgs n =` defines a recursive function `send_msgs`.
- `if n = 0 then ()` checks if the current value of `n` is 0, returning an empty event if true.
- `else (event[send]; sendmsgs (n-1))` recursively calls `sendmsgs` with `n-1` if `n` is not 0.

### (b) Typing Rules and Final Effect Approximations

- `A B` represents a typing judgment.
- `n ≥ 0 ⊢ (unit & $\Phi_{send msgs}'$)` is a constraint that must be satisfied for the function to type.

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

- A type derivation tree illustrates the structure of the type judgment, where each node represents a rule application.
- The tree shows how the types and effects are derived for the recursive function `send_msgs`.
- The tree includes deductions for the type and final effect approximations, as well as the constraints that must be satisfied.

---

**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.
finite/infinite effect pair \((\Phi^\mu, \Phi^\nu)\), under a typing environment where a total application of \(f\) has some finite/infinite effect pair \((\lambda x \in \Sigma^\mu X_\mu(x), \lambda x \in \Sigma^\nu X_\nu(x))\). \(X_\mu\) and \(X_\nu\) are finite and infinite predicate variables, respectively. Given this, the effect of a total application of the recursive function is then the effect pair \((\lambda x \in \Sigma^\mu q_\mu(x), \lambda x \in \Sigma^\nu q_\nu(x))\) where our type system requires that \(q_\mu = \mu X_\mu(x), \Phi^\mu(x)\) and \(q_\nu = \nu X_\nu(x), \Phi^\nu(x)\). 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 \(X_\mu\) (resp., \(X_\nu\)). The type system also generates constraints in other rules, such as the subtyping rules (S-QUAL, etc.) of the form \((\tau_1 \& \Phi_1) \ll (\tau_2 \& \Phi_2)\). In these cases, the type system requires that the finite (resp., infinite) effects \(\Phi^\mu_2\) (resp., \(\Phi^\nu_2\)) is approximated by the finite (resp., infinite) effects \(\Phi^\mu\) (resp., \(\Phi^\nu\)).

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 sendmsgs recursive function, whose overall type is given by the dependent-refinement type \(\tau_{\text{sendmsgs}}\), that constrains input \(n\) to be greater than or equal to 0. The overall effect \(\Phi_{\text{sendmsgs}}\) has two parts: the finite effect \(\Phi^\mu_{\text{sendmsgs}}\) and the infinite effect \(\Phi^\nu_{\text{sendmsgs}}\). These effect predicates involve predicate variables \(X_\mu\) and \(X_\nu\), quantified with a least and greatest fixpoint, respectively. Notice that \(X_\mu\) and \(X_\nu\) 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:

\[
\begin{align*}
\Phi^\mu_{\text{until-ready}} &= (\lambda x \in \text{Wait}^* \cdot \text{Ready}, \lambda x \in \text{Wait}^o) \\
\Phi^\mu_{\text{sendmsgs}} &= (\lambda x \in \text{Send}^o, \lambda x \in \text{Wait}^o) \\
\Phi^\mu_{\text{messenger}} &= (\lambda x \cdot \lambda x \in (\text{Ready} \cdot \text{Send}^o | \text{Wait}^o))
\end{align*}
\]

Intuitively, until-ready has finite behaviors that repeat \text{Wait} finitely many times followed by \text{Ready}. The infinite behaviors of until-ready are infinite repetition of \text{Wait}. \text{sendmsgs} has only finite behaviors, specifically, repetition of \text{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 \(\Phi^\mu\) of effect predicate \(\Phi\) provided that we can show that \(\forall x: \Phi^\mu(x) \Rightarrow \Phi(x)\). For sendmsgs, the sub-typing 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 how at two of the rules can be used for the example.
First, looking at the finite effects of sendmsgs, our rules allow us to show, for example, that \( \lambda x. x \in \text{Send} \) approximates \( \Phi^F_{\text{sendmsgs}} \) by using invariant predicates that over-approximate the least fix-\( \Phi^F_{\text{sendmsgs}} \) points. 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 \( \Phi^F_{\text{sendmsgs}} \). Our proof rule (\( Fp-LFP^* \)) lets us approximate this buried least fixpoint with \( \lambda(n,x).x \in \text{Send} \) by using the pre-fixpoint. In the first premise of the rule, we consider only the fixpoint and must show that when we substitute \( \lambda(n,x).x \in \text{Send} \) into the fixpoint formula, the result is approximated by \( \lambda(n,x).x \in \text{Send} \). In the second premise, we use this information, eliminating the fixpoint.

Next, looking at the infinite effects of sendmsgs, our proof system lets us show that the goal effect approximation \( \lambda x. \bot \) of \( \Phi^F'_{\text{sendmsgs}} \) 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 (\( vX(\tilde{x}), \tilde{y}(\tilde{t}) \)) by some predicate \( \neg p_1(\tilde{t}) \). 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 \( vX(\tilde{x}), T \land \tilde{y} \). In the sendmsgs example, we use the predicate \( \neg (n \geq 0) \) to approximate the GFP formula in \( \Phi^F'_{\text{sendmsgs}} \). What remains is the side-condition, where we use \( p_2 = n_1 \geq n_2 \geq 0 \) to witness that \( \neg p_1 \) over-approximates \( n \not\leq 0 \land \exists y.x \in \text{Send} \), \( y \land X_1(n - 1, y) \).

We treat this side-condition of witnessing predicates’ approximations itself as a judgment (denoted \( X_1(\tilde{x}) \); \( \neg p_1, T \land \tilde{y} \)) as well as an analogous least-fixpoint judgment (denoted \( X(\tilde{x}) \); \( p_1, p_2, \bot \land \tilde{y} \)) in another series of proof rules. These rules are inductively defined over \( \tilde{y} \), 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 (\( Ax\tilde{X} - \land \)), existential quantification (\( Ax\tilde{X} - \exists \)), 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 Appendix C.) 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 12 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 11 and 12: if 11 is empty, it moves everything from 12 to 11 and, otherwise, it dequeues by returning a pair of the dequeued item and the new queue (11′, 12). Here, \( a(x) \) is the number of \( a \)'s in \( x \). The temporal effect \( \Phi \) of main asserts that, when the program terminates, the number of enqueues plus the length of 12 is equal to the number of dequeues minus the length of 11, which is equal to the number of Tick's.

Higher-Order Functions. The second example shrinks 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., [24]). The effect \( \Phi \) asserts that shrink never terminates, and its infinite executions emit the event sequences (\( Shrink^{1/d} \)). 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 (\( Wait \)), accepts them (\( Accept \)) and dispatches them to an appropriate handler (\( Handle \)). 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 \( \ast \) 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 (\( (\tilde{X}' \cdot (\Sigma \setminus \{Accept\})^{npool + 1})^\omega \)), 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 \( L \) is shown in Fig. 3. Here, \( n, x, a \) are meta-variables ranging respectively over integers, term variables, and events. \( \Sigma \) represents a finite set of events. We write \( \tilde{x} \) for a finite sequence of variables and \( [\tilde{x}] \) for the length of \( \tilde{x} \). We also write \( e \) for the empty sequence. We use a meta-variable \( o \) (resp. \( \pi \)) to represent a finite (resp. infinite) sequence of events. We write \( o \cdot \pi \) (resp. \( o \cdot o' \)) for the concatenation of the finite \( o \) and the infinite \( \pi \) (resp. finite \( o' \)) sequences. \( op \) represents binary integer
We write \( fpv \) term and predicate variables. We write \( fv \) first-order fixpoint formulas in Appendix H. We write such as integers \( f \) as the equality on integers and sequences.

operators such as +, −, ×, =, and <. We assume that boolean and unit values are encoded as integers (e.g., true = 0 and false = 1).

We assume that expressions are simply-typed. An expression \( \text{ev}[a] \) raises the event \( a \). An expression if \( z \) then \( e_1 \) else \( e_2 \) reduces to \( e_1 \) if \( z = 0 \) and \( e_2 \) otherwise. We abbreviate let \( x = e_1 \) in \( e_2 \) as \( e_1;e_2 \) if \( x \) does not occur in \( e_2 \). An expression \( \text{rec}(f,\bar{x},e) \) represents a (possibly recursive) function \( f \) with the arguments \( \bar{x} \) (where \( |\bar{x}| \geq 1 \)) and the body \( e \). We assume that \( \text{rec}(f,\bar{x},e) \) 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. 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 \( L \). It is easy to extend our type system in Sec. 4 with these features (see, e.g., [23, 25]).

The operational semantics of \( L \) is defined by the set of inductive and coinductive rules for deriving judgments of the form \( e \Downarrow v \mathbin{\&} \omega \) and \( e \Downarrow \bot \mathbin{\&} \omega \). 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 \( \omega \). The latter is for non-terminating evaluations and means that the evaluation of \( e \) diverges producing the infinite sequence of events \( \omega \). The rules are analogous to the ones from [12] and described in Appendix A.

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. \( A(T) \) represents atomic formulas such as the equality on integers and sequences. \( f \) represents constants such as integers \( n \), the empty sequence \( e \), and singleton sequences \( a \) as well as functions such as the sequence concatenation and integer arithmetic operators. We write \( T \) and \( \bot \) respectively for tautology and contradiction. The formula \( \mu X : \bar{T}.\phi \) (resp. \( \nu X : \bar{T}.\phi \)) represents the least (resp. greatest) fixpoint (of the function \( LX : \bar{T}.\phi \)). We assume that \( X \in A(T) \). The variables \( X \) are free in \( \phi \). We sometimes omit sorts when they are obvious from the context. We define \( (\lambda X.\phi)(\bar{x}) \equiv [\bar{x}/X]\phi \) and write \( p_1 \Downarrow p_2 \) if \( \forall \bar{x}.p_1(\bar{x}) \Rightarrow p_2(\bar{x}) \) holds. We also write \( \neg\lambda X.\phi \) for \( \lambda X.\neg\phi \). We define the formal semantics of this first-order fixpoint formulas in Appendix H. We write \( \text{fv}(\phi) \) (resp. \( \text{fpv}(\phi) \)) for the set of free term (resp. predicate) variables in \( \phi \). We also define \( \text{fv}(\lambda X.\phi) \equiv \text{fv}(\phi) \setminus \{X\} \) and \( \text{fpv}(\lambda X.\phi) \equiv \text{fpv}(\phi) \).

4.2 Syntax of Types and Effects

The syntax of types and effects is shown in Fig. 5. An effect \( \Phi \) is a pair of predicates \( \lambda X . \Phi_p \) and \( \lambda X . \Sigma \Phi_v \), which may contain free term and predicate variables. We write \( \Phi^\tau \) (resp. \( \Phi^v \)) for \( \lambda X . \Sigma^\tau . \Phi_p \) (resp. \( \lambda X . \Sigma^v . \Phi_v \)). \( \Phi^\tau \) specifies a set of valid finite event sequences for terminating runs. On the other hand, \( \Phi^v \) specifies a set of valid infinite event sequences for non-terminating runs. We define the concatenation \( \Phi_1 \cdot \Phi_2 \) of effects \( \Phi_1 \) and \( \Phi_2 \) as follows.1

\[
(\lambda X . \Sigma^\tau . \Phi_p, \lambda X . \Sigma^v . \Phi_v) \cdot (\lambda X . \Sigma^\tau . \Phi'_p, \lambda X . \Sigma^v . \Phi'_v) = (\lambda X . \Sigma^\tau . (\Phi_p \cup \Phi'_p), \lambda X . \Sigma^v . (\Phi_v \cup \Phi'_v))
\]

We also define a special effect \( \Phi_{\bot} \) as \( (\lambda X . \Sigma^\tau . \Phi_p, \lambda X . \Sigma^v . \Phi_v) \). Note that \( \Phi_{\bot} \) is an identity with respect to \( \cdot \), that is, \( \Phi_{\bot} \cdot \Phi = \Phi \cdot \Phi_{\bot} = \Phi \). We often abbreviate \( (\tau \& \Phi_{\bot}) \) as \( \tau \).

An effect qualified type is of the form \( (\tau \& \Phi) \) where \( \tau \) is a dependent refinement type (described below) and \( \Phi \) is an effect. Roughly, the qualified type \( (\tau \& \Phi) \) is the type of expressions \( e \) such that, 1.) for all terminating run \( e \Downarrow v \mathbin{\&} \omega \), \( v \) conforms to the type \( \tau \) and \( \omega \) satisfies \( \Phi \), and 2.) for all non-terminating run \( e \Downarrow \bot \mathbin{\&} \omega \), \( \omega \) satisfies \( \Phi^v \). Sec. 4.3 formally defines the semantics of the qualified types.2

For example, \( (\tau \& \Phi) \) is the type of expressions that take an argument \( x \) of the type \( \tau \) and behave according to the return type \( \sigma \). Note that the scope of \( x \) is within \( \sigma \), and hence effects in \( \sigma \) can depend on the argument \( x \). For example, \( (\tau \& \Phi) \rightarrow (\tau \& \Phi) \) is the type of expressions that exhibit event sequences conforming to \( \Phi \) when evaluated, and cause event sequences conforming to \( \Phi \) when applied to some integer argument \( x \). We write \( (\bar{x}, \tau) \) for \( (\tau \& \Phi) \) when \( \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \) are all arguments. We note that a type of such a form can be given to a recursive function \( \text{rec}(f,\bar{x},e) \) where \( \bar{x} = [\bar{x}] \) as partial applications to \( f \), i.e., applying less than \( [\bar{x}] \) many arguments do not raise any events. We abbreviate \( (\tau \& \Phi) \rightarrow (\tau \& \Phi) \) as \( \tau \rightarrow \tau \) for \( x \) does not occur in \( \sigma \).

A type environment \( \Gamma \) is a sequence of variable bindings \( x : \tau \). We define \( \Gamma(\bar{x}) \equiv \tau \) if \( x : \tau \in \Gamma \). We abbreviate \( \Gamma, \nu : \{v | \phi\} \) as \( \Gamma, \phi \) if \( v \not\in \text{fv}(\phi) \) and \( v \) never occurs elsewhere. Note that type bindings in type environments and arguments of function types are of the form \( (\tau \& \Phi) \) instead of \( (\tau : \sigma) \). This is because the target language \( L \) is call-by-value, and hence variables are always bound to values whose evaluation never exhibits temporal effects.

We define auxiliary functions \( \text{stv}(\sigma), \text{fv}(\sigma), \) and \( \text{fpv}(\sigma) \). \( \text{stv}(\sigma) \) represents the simple type corresponding to the qualified type \( \sigma \). \( \text{fv}(\sigma) \) (resp. \( \text{fpv}(\sigma) \)) represents the set of free term (resp. predicate) variables that occur in \( \sigma \). The definitions are standard and deferred to Appendix B. We extend the notions to type environments and define \( \text{stv}(\Gamma), \text{fv}(\Gamma) \), and \( \text{fpv}(\Gamma) \) in the obvious way.

---

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

2Readers 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.
We remark that our type and effects are essentially the extension of the types from the previous work on dependent-refinement type systems [12, 19, 22, 23, 27, 29] with dependent temporal effects, which are the effect of the expression 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.

![Figure 6. Semantic typing](image)

We remark that our type and effects are essentially the extension of the types from the previous work on dependent-refinement type systems [12, 19, 22, 23, 27, 29] with dependent temporal effects, which are the effect of the expression 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.

4.4 Typing Rules

![Figure 7. Typing and subtyping rules](image)

**LICS ’18, July 9–12, 2018, Oxford, United Kingdom**
\[\begin{align*}
&\vdash \psi' \quad \text{FP-VALID} \quad \vdash \left(\lambda x.\psi'/X\right)\psi \Rightarrow \psi' \quad \text{FP-LFP}^\sim \\
&\vdash \psi' \Rightarrow \left(\lambda x.\psi'/X\right)\psi \quad \text{FP-LFP}^+ \quad \vdash \left(\mu X.\psi\right)(\bar{t}) \quad \text{FP-LFP}^-
\end{align*}\]

Figure 8. Proof rules for the fixpoint logic

the effect of the function application. Note here that the variable \(x\) may occur in the latent effect, which is substituted by \(\bar{t}_i\) 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 \([\Gamma]\) and \(\Gamma \vdash \phi\) are defined by:

\[\begin{align*}
[\emptyset] &\triangleq \top \\
[\Gamma, x : (y:\phi)] &\triangleq [\Gamma] \land [x/y]\phi \\
[\Gamma \vdash \phi] &\triangleq [\Gamma] \\
[\Gamma, x : (y:\tau) \rightarrow \sigma] &\triangleq [\Gamma]
\end{align*}\]

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 \(\tau_1\) and \(\tau_2\) are in the subtyping relationship. Further, it checks that the left effect \(\Phi_1\) is a subeffect of the right effect \(\Phi_2\). The subeffecting relation checks that the finite (resp. infinite) part of \(\Phi_1\) logically implies the finite (resp. infinite) part of \(\Phi_2\), under the assertions implied by the type environment \(\Gamma\). For example, in the typing of messenger from Sec. 2, a subtyping judgment \(\Gamma \vdash (\tau \rightarrow \Phi_2)\) : \((\sigma \triangleq \Phi_2)\) is discharged where \(\Gamma = n : x.\{x \mid x \geq 0\}\), \(\tau\) is the refinement integer type obtained for \(\Phi_2\), and \(\Phi_1\) and \(\Phi_2\) are from Fig. 1. S-QUAL checks the subtyping by asserting the validity of \(n \geq 0 \Rightarrow \forall x \in \Sigma^*.\Phi_2\) \(\Rightarrow \Phi_1\) \(\vdash_{\text{sendmsgs}}(x)\) and \(n \geq 0 \Rightarrow \forall x \in \Sigma^*.\Phi_2\) \(\Rightarrow \Phi_1\) \(\vdash_{\text{sendmsgs}}(x)\). Sec. 5 shows the deductive system for solving such predicate fixpoint logic constraints.

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

\textbf{Theorem 4.1.} If \(\Gamma \vdash e : \sigma\), then \(e \in \rho(\Gamma) \Rightarrow \rho(\sigma)\) for any predicate substitution \(\rho\) with \(\text{dom}(\rho) = \text{fpv}(\Gamma) \cup \text{fpv}(\sigma)\).

We remark that the soundness holds for any background first-order theory supporting basic integer arithmetic (i.e., those in \(L\)) 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, 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 \(\phi\) is valid. The derivation rules for \(\vdash \phi\) are shown in Fig. 8. There, metavariable \(\psi\) ranges over formulas not containing fixpoint formulas (i.e., those of the form \((\mu X.\phi)(\bar{t})\) and \((\nu X.\phi)(\bar{t})\)). The formula \(\text{nff}(\phi)\) is the negation normal form of \(\phi\), and \(\vdash WF(p)\) means that the predicate \(p = \lambda x.\phi\) is well-founded, that is, the arity of \(p\) is 2 \(\times n\) for some \(n\) and there is no infinite sequence \(t_1, t_2, \ldots\) such that \([t_i] = n \text{ and } p([t_i], [t_{i+1}])\) holds for all \(i \geq 1. C^+(\text{resp. } C^-)\) 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. FP-LFP-\(\sim\) over-approximates a least fixpoint \(\mu X.\psi\) that occurs in a negative position with a pre-fixpoint \(\lambda X.\psi^*\) of the function \(\lambda X.\lambda x.\phi\). Note here that \(\psi\) and \(\psi^*\) do not contain fixpoints. An example of FP-LFP-\(\sim\) can be seen in Sec. 2, where we discuss proof subtre A. Meanwhile, FP-GFP\(\sim\) is a dual rule and it under-approximates a greatest fixpoint \(\nu X.\psi\) that occurs in a positive position with a post-fixpoint \(\lambda X.\lambda x.\phi\).

By contrast, FP-LFP\(\sim\) under-approximates a least fixpoint that occurs in a positive position with a predicate \(p_1\). Here, the auxiliary judgment \(X(\bar{t}); \vdash (\rho p_1; p_2; \psi' \vdash \psi)\) is used to check that the well-founded predicate \(p_2\) witnesses that \(p_1\) under-approximates the least fixpoint \(\mu X.\psi\). In a dual manner, FP-GFP\(\sim\) over-approximates a greatest fixpoint that occurs in a negative position with a predicate \(-p_1\). The auxiliary judgment \(X(\bar{t}); \vdash (\rho p_1; p_2; \psi' \vdash \psi)\) checks that the well-founded predicate \(p_2\) witnesses that \(-p_1\) over-approximates the greatest fixpoint \(\nu X.\psi\). An example of FP-GFP\(\sim\) can be seen in Sec. 2, where we discuss proof subtre 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 \(X(\bar{t}); \vdash (\rho p_1; p_2; \psi' \vdash \psi) \land X(\bar{t}); \vdash (\rho p_1; p_2; \psi' \vdash \psi)\) defined in Fig. 9. There, we maintain the invariants that \(\psi\) is in the negation normal form, \(\psi\) does not contain \(X\), and \(X\) may occur only positively in \(\psi\). The rules let us then manipulate the judgments to further reduce to predicate reasoning. The rules APx\(\sim\)-\(\wedge\) and APx\(\sim\)-\(\lor\) are similar to standard ones for first-order logic. APx\(\sim\)-\(\rightarrow\) splits a judgment with the succedent of the form \(\psi_1 \lor \psi_2\) into two judgments: one with the succedent \(\psi_1\) and the antecedent conjuncted with \(\psi_2\) and the other with the succedent \(\psi_2\) and the antecedent conjuncted with \(\psi_1\) for some \(\psi_1\) and \(\psi_2\) such that \(\psi_1 \lor \psi_2\) holds provided the original antecedent does. APx\(\sim\)-\(\exists\) generalizes APx\(\sim\)-\(\rightarrow\) to judgments with the succedent of the form.
A Fixpoint Logic and Dependent Effects for Temporal Property Verification

Theorem 5.2. If $\phi$, then $\phi$.

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 Appendix H 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, 18, 19, 22-24, 30, 31], termination [13, 27], non-termination [3, 14], and properties expressed in linear $\mu$-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, 17], whereas the termination verification method of [13] and linear $\mu$-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 class of properties that are not addressed in this paper are branching properties, such as those expressible in the branching $\mu$-calculus. Sound and complete methods for the class exist for well-typed finite-data higher-order programs (i.e., higher-order recursion schemes) [8, 17]. For infinite-data higher-order programs, a recent work by Unno et al. [25] 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, 20]. 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, 20] only allow ($\omega$)-regular sets
of event sequences, and [12] (without oracles) always assigns $\nu^\omega$ 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 $\mu$-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 [21], 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, 28]. 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 formulas on event sequences and program values. While fixpoint logics such as $\mu$-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 [21], 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, 28]. 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.

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

infinite sequence $\pi$ of events. Note that the derivation rules for $e \downarrow v \& \omega$ are defined inductively, while those for $e \uparrow \bot \& \pi$ are defined coinductively. The rule RT-EVENT evaluates $\text{ev}[a]$ to 0, raising the event $a$.

### B Auxiliary Functions

Fig. 11 defines the auxiliary functions $\text{sty}(\sigma)$, $\text{fv}(\sigma)$, and $\text{fpv}(\sigma)$. $\text{sty}(\sigma)$ represents the simple type corresponding to the qualified type $\sigma$. $\text{fv}(\sigma)$ (resp. $\text{fpv}(\sigma)$) represents the set of free term (resp. predicate) variables that occur in $\sigma$. We extend the notions to type environments and define $\text{sty}(\Gamma)$, $\text{fv}(\Gamma)$, and $\text{fpv}(\Gamma)$ in the obvious way.

### C Types for Other Examples

The types of the functions in the examples are listed in Figures 12, 13, and 14. There, $\tau_{\text{foo}}$ represents a type of a function $\text{foo}$, $\Phi_{\text{foo}}$ represents the type of $\text{foo}$, and $\Phi'_{\text{foo}}$ represents an approximation of $\Phi_{\text{foo}}$ sufficient for type checking.

### D Proof of Theorem 5.2

We first prove Lemma 5.1 which states the correctness of the fix-point approximation rules. The following are lemmas used to show Lemma 5.1.

**Lemma D.1.** If $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$ and $\models \phi \Rightarrow (\mu X \Psi_2)(\bar{x})$, then $\models \phi \Rightarrow ((\mu X \Psi_2)(\bar{x}) \land (\mu X \Psi_2)(\bar{x})$.

**Lemma D.2.** If $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$ and $\models \phi \Rightarrow (\lambda x \Psi x)$, then $\models \phi \Rightarrow ((\mu X \Psi)(\bar{x}) \land (\mu X \Psi)(\bar{x})$.

**Lemma D.3.** If $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$ and $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$, then $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$.

**Lemma D.4.** If $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$ and $\models \phi \Rightarrow (\lambda x \Psi x)$, then $\models \phi \Rightarrow (\mu X \Psi)(\bar{x})$.

**Lemma D.5.** If $\models (\lambda x \Psi x)(\bar{x})$, then $\models (\mu X \Psi)(\bar{x}) = (\mu X \Psi)(\bar{x})$.

**Lemma D.6.** If $\models (\lambda x \Psi x)(\bar{x})$, then $\models (\mu X \Psi)(\bar{x}) = (\mu X \Psi)(\bar{x})$.

**Lemma D.7.** If $\models (\lambda x \Psi x)(\bar{x})$, then $\models (\mu X \Psi)(\bar{x}) = (\mu X \Psi)(\bar{x})$.

**Lemma D.8.** If $\models (\lambda x \Psi x)(\bar{x})$, then $\models (\mu X \Psi)(\bar{x}) = (\mu X \Psi)(\bar{x})$.

**Proof of Theorem 5.1.**

1. By induction on the derivation of $X(\bar{x})$: $p_1; p_2; \Psi' \downarrow \Psi$.
   - **Case Apx²-BASE:** We have $\models p_1(\bar{x}) \land \Psi' \Rightarrow \Psi$. Therefore, we get $\models p_1(\bar{x}) \land \neg \Psi' \Rightarrow \Psi$. Because $X$ in $\neg \Psi' \lor \Psi$ is a free predicate variable, by substituting $\mu X(\bar{x}) \lnot \Psi' \lor \Psi$ to $X$ we get $\models p_1(\bar{x}) \Leftarrow (\mu X \neg \Psi' \lor X) \Rightarrow \neg \Psi' \lor \Psi$. Because $\mu X(\bar{x}) \lnot \Psi' \lor X \Rightarrow \Pi X \lnot \Psi' \lor \Psi$ (by $\nu$), we obtain $\models p_1(\bar{x}) = (\mu X \nu \Psi' \lor \Psi)$.
   - **Case Apx²-Rec:** We have
     - $\Psi = X(\bar{x})$
     - $\models (p_1(\bar{x}) \land \Psi') = (p_1(\bar{x}) \land \nu \Psi)$. We define $\psi_i$ as following:
       $$\psi_0 = \bot$$
       $$\psi_{i+1} = \neg \psi' \land \Pi \psi_i$$
       By $\models p_1(\bar{x}) \land \Psi' \Rightarrow p_1(\bar{x})$, we get $\models p_1(\bar{x}) \land \Psi' \Rightarrow \neg \psi' \lor X(\bar{x}) \Rightarrow p_1(\bar{x})$. But $p_1(\bar{x}) \land \Psi' \Rightarrow p_2(\bar{x}, i)$, so $\models p(\bar{x}) \land \Psi' \Rightarrow \Pi \psi_i$ holds. Therefore we get $\models p_1(\bar{x}) \land \Psi' \Rightarrow \Pi \psi_i$. By Lemma 4.8, we obtain $\models p_1(\bar{x}) \land \Psi' \Rightarrow (\mu X \nu \Psi)(\bar{x})$.

Using the auxiliary functions defined in the text and the results from the previous sections, we can prove the correctness of the fix-point approximation rules.
\[ r_{\text{aux}} = (l : \text{int list} \times \text{int list}) \rightarrow (a : \text{int list} \times \text{int list}) \rightarrow ((l' \ | \ l'') = |l| + |a|) \& \Phi_{\text{aux}} \]
\[ \Phi_{\text{aux}} = \lambda x. (\mu X_p(l, a, x). l = 0 \wedge x = e \vee l \neq 0 \& \exists y, l', a' . x = \text{Tick} \cdot y \wedge |l'| = |l| - 1 \wedge |a'| = |a| + 1 \wedge X_p(l', a', y))(l, a, x) \]
\[ \Phi'_{\text{aux}} = \lambda x. (\nu X_p(l, a, x). l = 0 \wedge x = e \vee l \neq 0 \& \exists y, l', a' . x = \text{Tick} \cdot y \wedge |l'| = |l| - 1 \wedge |a'| = |a| + 1 \wedge X_p(l', a', y))(l, a, x) \]
\[ \tau_{\text{rev}} = (l : \text{int list}) \rightarrow ((u | u = |l|) \& \Phi_{\text{rev}}) \]
\[ \Phi_{\text{rev}} = [0/a] \Phi_{\text{aux}} \]
\[ \Phi'_{\text{rev}} = (\lambda x. e \in \text{Tick} \cdot \lambda x. l) \]
\[ r_{\text{is_empty}} = (l_1, l_2) : \text{int list} \times \text{int list} \rightarrow ((u | u = (l_1 = 0) \& (l_2 = 0)) \& \Phi_{\text{is_empty}}) \]
\[ \Phi_{\text{is_empty}} = (\lambda x. (\mu X_p((l_1, l_2), x). x = e))(l_1, l_2), (\lambda x. (\nu X_p((l_1, l_2), x). x = e))(l_1, l_2) \]
\[ \Phi'_{\text{is_empty}} = \Phi_{\text{val}} \]
\[ r_{\text{enqueue}} = (e : \text{int}) \rightarrow (l_1, l_2) : \text{int list} \times \text{int list} \rightarrow (((l_1, l'_1) | l'_1 = 1 + |l_2|) \& \Phi_{\text{enqueue}}) \]
\[ \Phi_{\text{enqueue}} = (\lambda x. (\mu X_p(e, (l_1, l_2), x). x = \text{Enq}(e, (l_1, l_2), x), (\lambda x. (\nu X_p(e, (l_1, l_2), x). x = e))(l_1, l_2), x) \]
\[ \Phi'_{\text{enqueue}} = (\lambda x. x = \text{Enq}(e, l_1, l_2) \& \lambda x. l) \]
\[ r_{\text{dequeue}} = (l_1, l_2) : \text{int list} \times \text{int list} \rightarrow (((e, (l'_1, l'_2)) | l'_1 = 0 \& |l'_2| = 1 + |l_2|) \& \Phi_{\text{dequeue}}) \]
\[ \Phi_{\text{dequeue}} = (\lambda x. (\mu X_p((l_1, l_2), x). l = 0 \& \exists y_1, y_2 . x = y_1 \cdot y_2 \wedge |l_2/l| \Phi_{\text{dequeue}}(y_1) \wedge X_p(l_2, 0, y_2) \vee l \neq 0 \& x = \text{Deq}(x_1, x_2, l_2, y_1, y_2, x = y_1 \cdot y_2 \wedge \Phi_{\text{dequeue}}(y_1) \wedge X_p(l_2, 0, y_2))(l_1, l_2), x) \]
\[ \Phi'_{\text{dequeue}} = (\lambda x = \text{Deq}(x, l_1, l_2) \& \lambda x. l) \]
\[ r_{\text{main}} = (l_1, l_2) : \text{int list} \times \text{int list} \rightarrow (\text{unit} \& \Phi_{\text{main}}) \]
\[ \Phi_{\text{main}} = \lambda x. (\mu X_p((l_1, l_2), x). (\exists y_1, y_2 . x = y_1 \cdot y_2 \wedge |l_2/l| \Phi_{\text{dequeue}}(y_1) \wedge X_p(l_2, 0, y_2) \vee l = 0 \wedge l_2 = 0 \& x = e) \vee (l = 0 \& l_2 \neq 0 \& \exists y_1, y_2 . x = y_1 \cdot y_2 \& \Phi_{\text{dequeue}}(y_1) \wedge X_p(l_2, 0, y_2) \vee l \neq 0 \& \exists y_1, y_2 . x = y_1 \cdot y_2 \& \Phi_{\text{dequeue}}(y_1) \wedge X_p(l_2, 0, y_2))(l_1, l_2), x) \]
\[ \Phi'_{\text{main}} = \lambda x. x \neq \lambda x. l \]

**Figure 12.** Types and effects for the Amortized Complexity example in Figure 2.

- **Case Apx^\Phi_{\text{aux}}-\land** We have:
  - \( \psi = \psi_1 \land \psi_2 \)
  - \( X(\tilde{x}); p_1; p_2; \psi' \lor \psi_1 \)
  - \( X(\tilde{x}); p_1; p_2; \psi' \lor \psi_2 \)
  - By L.H. we obtain
    - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor \psi_1)(\tilde{x}) \)
    - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor \psi_2)(\tilde{x}) \)
  - By Lemma D.1, we obtain \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor \psi_1 \land \psi_2)(\tilde{x}) \)

- **Case Apx^\Phi_{\text{aux}}-\lor** We have:
  - \( \psi = \psi_1 \lor \psi_2 \)
  - \( |p_1(\tilde{x}) \land \psi' \Rightarrow (\psi_1' \lor \psi_2' \land \neg \psi'_1 \lor \neg \psi'_2)(\tilde{x}) \)
  - By L.H., we get
    - \( |p_1(\tilde{x}) \Rightarrow (\mu X(\tilde{x}), \neg \psi' \lor (\psi_1 \lor \neg \psi'_1 \land \neg \psi'_2)(\tilde{x}) \).
    - Therefore, we have \( |p_1(\tilde{x}) \Rightarrow (\mu X(\tilde{x}), \neg \psi' \lor (\psi_1 \lor \neg \psi'_1 \land \neg \psi'_2 \lor \neg \psi'_1 \lor \psi'_2)(\tilde{x}) \).
    - Because \( |(\neg \psi' \lor (\psi_1 \lor \psi'_1 \land (\psi_2 \lor \neg \psi'_2 \lor \psi'_2))(\tilde{x}) \Rightarrow (p_1(\tilde{x}) \Rightarrow \neg \psi' \lor (\psi_1 \lor \psi'_1 \land (\psi_2 \lor \neg \psi'_2 \lor \psi'_2))(\tilde{x}) \)
    - By Lemma D.2, we obtain \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor (\psi_1 \lor \psi'_1 \land (\psi_2 \lor \neg \psi'_2 \lor \psi'_2))(\tilde{x}) \).

- **Case Apx^\Phi_{\text{aux}}-\forall** We have:
  - \( \forall \equiv \exists x . \psi_1 \)
  - \( X(\tilde{x}); p_1; p_2; \psi' \lor \exists \tilde{x}', \psi'' \)
  - \( \neg \psi'' \lor \exists \tilde{x}'. \psi'' \)
  - By L.H., we get
    - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor \exists \tilde{x}', \psi'' \)
    - By Lemma D.3 and that \( \forall \psi \lor \exists \tilde{x}'. \psi'' \), we have
      - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \neg \psi' \lor \exists \tilde{x}', \psi'' \)

- **Case Apx^\Phi_{\text{aux}}-\exists** We have:
  - \( \exists \equiv \exists x . \psi_1 \)
  - \( |(p_1(\tilde{x}) \land \psi') \Rightarrow \exists \tilde{x}'. \psi'' \)
  - \( \neg \psi' \lor \forall \tilde{x}'. \psi'' \)
  - \( \neg \psi' \lor \forall \tilde{x}'. \psi'' \)
  - By L.H., we obtain
    - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \psi' \lor \exists \tilde{x}'. \psi'' \Rightarrow [\tilde{x}', \psi_1] \)
    - By Lemma D.4, we have
      - \( |p_1(\tilde{x}) = (\mu X(\tilde{x}), \psi' \lor \exists \tilde{x}'. \psi'' \Rightarrow [\tilde{x}', \psi_1] \)

2. By induction on the derivation of \( X(\tilde{x}); p_1; p_2; \psi' \lor \psi \).

- **Case Apx^\Phi_{\text{aux}}-Base** We have \( |p_1(\tilde{x}) \land \psi' \Rightarrow \neg \psi \). We then get \( |p_1(\tilde{x}) \Rightarrow \neg \psi' \lor \neg \psi \). By contraposition, we get \( |\neg \psi' \lor \neg \psi \Rightarrow \neg \psi \). Because \( X \in \psi' \lor \psi \) is a free predicate variable, by substituting \( \nu X(\tilde{x}), \psi' \lor \psi \) to \( X \)
\[\tau_{\text{zoom}} = \text{unit} \rightarrow (\text{unit} \& \Phi_{\text{zoom}})\]
\[\Phi^\mu_{\text{zoom}} = \lambda x.(\mu X(x).\exists y. x = \text{Zoom} \cdot y \land X(y))(x)\]
\[\Phi^\nu_{\text{zoom}} = \lambda x.(\nu X(x).\exists y. x = \text{Zoom} \cdot y \land X(y))(x)\]
\[\Phi'^\mu_{\text{zoom}} = \lambda x.\bot\]
\[\Phi'^\nu_{\text{zoom}} = \lambda x. x \in \text{Zoom}^{\omega}\]

approximated by our deductive system with:
\[p_1 = \lambda ((), x) \notin \text{Zoom}^{\omega}\]
\[p_2 = \lambda (((), y) \exists x_1, y_1 \in \Sigma^\star, c_1, c_2 \in \Sigma, x_2, y_2 \in \Sigma^\omega. x = x_1 \cdot c_1 \cdot x_2 \land y = y_1 \cdot c_2 \cdot y_2 \land |x_1| > |y_1| \geq 0\]

\[\tau_{\text{shrink}} = (t : \{t \mid t \geq 0\}) \rightarrow (f : \text{unit} \rightarrow \{u \mid u = t\}) \rightarrow (d : \{d \mid d > 0 \land t \mod d = 0\}) \rightarrow (\text{unit} \& \Phi_{\text{shrink}})\]
\[\Phi^\mu_{\text{shrink}} = \lambda x.(\mu X(t, d, x). (t \leq 0 \land \Phi^\mu_{\text{zoom}}(x)) \lor (t > 0 \land \exists y. x = \text{Shrink} \cdot y \land X(y)(t - d, d, y)))(t, d, x)\]
\[\Phi^\nu_{\text{shrink}} = \lambda x.(\nu X(t, d, x). (t \leq 0 \land \Phi^\nu_{\text{zoom}}(x)) \lor (t > 0 \land \exists y. x = \text{Shrink} \cdot y \land X(y)(t - d, d, y)))(t, d, x)\]
\[\Phi'^\mu_{\text{shrink}} = \lambda x.\bot\]
\[\Phi'^\nu_{\text{shrink}} = \lambda x. x \in \text{Shrink}^{l/d} \cdot \text{Zoom}^{\omega}\]

approximated by our deductive system with:
\[p_1 = \lambda x. x \notin \text{Shrink}^{l/d} \cdot \text{Zoom}^{\omega}\]
\[p_2 = \lambda (t_1, d_1, x_1, x_2, x_3, x_4, d_2, d_3, d_4, t_2) t_1 > t_2 \geq 0\]

\[\tau_{\text{shrinker}} = (t : \{t \mid t > 0\}) \rightarrow (d : \{d \mid d > 0 \land t \mod d = 0\}) \rightarrow (\text{unit} \& \Phi_{\text{shrinker}})\]
\[\Phi^\mu_{\text{shrinker}} = \lambda x.(\mu X(t, d, x). \Phi^\mu_{\text{shrinker}}(x))(t, d, x)\]
\[\Phi^\nu_{\text{shrinker}} = \lambda x.(\nu X(t, d, x). \Phi^\nu_{\text{shrinker}}(x))(t, d, x)\]
\[\Phi'^\mu_{\text{shrinker}} = \lambda x.\Phi'^\mu_{\text{shrinker}}(x)\]
\[\Phi'^\nu_{\text{shrinker}} = \lambda x.\Phi'^\nu_{\text{shrinker}}(x)\]

Figure 13. Types and effects for the Shrinker example in Figure 2.

\[\tau_{\text{listener}} = (\text{npool} : \{v \mid v \geq 0\}) \rightarrow (\text{pend} : \{v \mid v \geq 0\}) \rightarrow (\text{unit} \& \Phi_{\text{listener}})\]
\[\Phi^\mu_{\text{listener}} = \lambda x.(\mu X(\text{npool}, \text{pend}, x). (\text{pend} < \text{npool} \land \exists y. x = \text{Accept} \cdot y \land X(\text{npool}, \text{pend} + 1, y) \lor (\text{pend} > 0 \land \exists y. x = \text{Handle} \cdot y \land X(\text{npool}, \text{pend} - 1, y) \lor (\text{pend} \leq 0 \land \exists y. x = \text{Wait} \cdot y \land X(\text{npool}, \text{pend}, y)\} (\text{npool}, \text{pend}, x)\]
\[\Phi^\nu_{\text{listener}} = \lambda x.(\nu X(\text{npool}, \text{pend}, x). (\text{pend} < \text{npool} \land \exists y. x = \text{Accept} \cdot y \land X(\text{npool}, \text{pend} + 1, y) \lor (\text{pend} > 0 \land \exists y. x = \text{Handle} \cdot y \land X(\text{npool}, \text{pend} - 1, y) \lor (\text{pend} \leq 0 \land \exists y. x = \text{Wait} \cdot y \land X(\text{npool}, \text{pend}, y)\} (\text{npool}, \text{pend}, x)\]
\[\Phi'^\mu_{\text{listener}} = \lambda x.\bot\]
\[\Phi'^\nu_{\text{listener}} = \lambda x. x \in (\Sigma^\star \cdot (\Sigma \setminus \text{Accept} \cdot \text{npool} \cdot \text{pend} + 1)^{\omega}) \Rightarrow x \in (\Sigma^\star \cdot \text{Wait}^{\omega})\]

approximated by our deductive system with:
\[p_1 = \lambda (\text{npool}, \text{pend}, x). \neg (x \in (\Sigma^\star \cdot (\Sigma \setminus \text{Accept} \cdot \text{npool} \cdot \text{pend} + 1)^{\omega}) \Rightarrow x \in (\Sigma^\star \cdot \text{Wait}^{\omega})\]
\[p_2 = \lambda (\text{npool}, \text{pend}, 1, x, \text{npool} + 2, \text{pend}, y) \exists x_1, y_1 \in \Sigma^\star, x_2, y_2 \in \Sigma^\omega. x = x_1 \cdot \text{Wait} \cdot x_2 \land y = y_1 \cdot \text{Wait} \cdot y_2 \land |x_1| > |y_1| \geq 0\]

\[\tau_{\text{server}} = (\text{npool} : \{v \mid v \geq 0\}) \rightarrow (\text{unit} \& \Phi_{\text{server}})\]
\[\Phi^\mu_{\text{server}} = \lambda x.(\mu X(\text{npool}, x).[0/\text{pend}][\Phi'^\mu_{\text{listener}}(x)](\text{npool}, x)\]
\[\Phi^\nu_{\text{server}} = \lambda x.(\nu X(\text{npool}, x).[0/\text{pend}][\Phi'^\nu_{\text{listener}}(x)](\text{npool}, x)\]
\[\Phi'^\mu_{\text{server}} = \lambda x.\bot\]
\[\Phi'^\nu_{\text{server}} = \lambda x. [0/\text{pend}][\Phi'^\nu_{\text{server}}(x)\]

Figure 14. Types and effects for the server example in Figure 2.
we obtain $\models [\nu X(\bar{x}).(\nu' \land \psi')] (\psi' \land \psi) \Rightarrow \neg p_1(\bar{x})$. Because $[\nu X(\bar{x}).(\nu' \land \psi')] (\psi' \land \psi) \iff (\nu X(\bar{x}).(\nu' \land \psi') (\bar{x}))$, we obtain $\neg p_1(\bar{x})$.

- **Case Apx'-Rec**: We have
  - $\emptyset \models X(\bar{f})$
  - $p_1(\bar{f}) \land \nu' \Rightarrow p_1(\bar{f}) \land p_2(\bar{x}, \bar{f})$

In a similar to Apx'-Rec, we have $\models p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi_i$. We define $\psi'_0 = \neg \psi_i$, that is

$$\psi'_0 = \top$$

$$\exists i.\psi'_i$$

By $\models p_1(\bar{x}) \land \nu' \Rightarrow \neg \bigwedge_{i=0}^{\infty} \psi'_i$. By Lemma E.5, we obtain $\models p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi'_i$. Therefore, we have $\models p_1(\bar{x}) \land \nu' \Rightarrow \neg \bigwedge_{i=0}^{\infty} \psi'_i$. So, we obtain $\models p_1(\bar{x}) \land \nu' \Rightarrow \neg \bigwedge_{i=0}^{\infty} \psi'_i$. By Lemma E.5, we get $\models p_1(\bar{x}) \land \nu' \Rightarrow (\nu X(\bar{x}).(\nu' \land X(\bar{f})))$. Therefore, $\models \nu' \land (\nu X(\bar{x}).(\nu' \land X(\bar{f}))) \Rightarrow p_1(\bar{x})$ holds. Finally, we obtain $\models (\nu X(\bar{x}).(\nu' \land X(\bar{f}))) \Rightarrow p_1(\bar{x})$.

- **Case Apx'-A**: We have
  - $\emptyset \models X(\bar{f})$
  - $p_1(\bar{f}) \land \nu' \Rightarrow (\psi'_1 \lor \psi'_2)$
  - $\nu X(\bar{f}) \subseteq \bar{x}$
  - $\emptyset \not\models \nu' X(\bar{f})$
  - $X(\bar{x}) \land p_1; p_2; \psi'_1 \land \psi'_2$
  - $i = 1, 2$

By I.H., we obtain $\models (\nu X(\bar{x}).(\nu' \land \psi'_1 \land \psi'_2)) \Rightarrow p_1(\bar{x})$ for $i = 1, 2$. Therefore, by Lemma D.5, we get $\models (\nu X(\bar{x}).(\nu' \land ((\psi'_1 \land \psi'_1) \lor (\psi'_2 \land \psi'_2)))) \Rightarrow p_1(\bar{x})$. Therefore, we have $\models (\nu X(\bar{x}).(\nu' \land (((\psi'_1 \land \psi'_1) \lor (\psi'_2 \land \psi'_2)) \land (\psi'_1 \land \psi'_2))) \Rightarrow p_1(\bar{x})$.

Figure 15. Type derivation for until._ready.

\[ \begin{array}{ll}
\text{until ready} : \tau \vdash \text{ev[R]}(\bar{x}) & \text{until ready} : \tau \vdash \text{ev[W]}(\bar{x}) \rightarrow \text{until ready}(\bar{x}) \rightarrow (\text{unit} \land \Phi'_{\text{until ready}})\\
\text{until ready} : \tau \vdash \text{if z then (ev[R]() else (ev[W]() (unit & \Phi'_{\text{until ready}}))))} & \vdash (\text{unit} \land \Phi'_{\text{until ready}}) \Rightarrow (\text{unit} \land \Phi'_{\text{until ready}})\\
\end{array} \]

\[ \begin{array}{ll}
\models p_1(\bar{x}) \Rightarrow x \in W' \land R & \models p_1(\bar{x}) \Rightarrow x \in W' \land R\\
A : \models (p_1(\bar{x}) \land x \neq W \land x' \land x') \Rightarrow (p_1(\bar{x}) \land x \neq W \land x') & \vdash (\neg x = W \land x' \land x') \Rightarrow (p_1(\bar{x}) \land x = W \land x')\\
X_0(\bar{x}); p_1; p_2; x = W \land x' \land x' & X_0(\bar{x}); p_1; p_2; x = W \land x' \land x'\\
& X_0(\bar{x}); p_1; p_2; x = W \land x' \land x'\\
B : \vdash (\nu x(\bar{x}) \exists y. x \models W \land x \land x(\bar{x}))(\bar{x}) \Rightarrow x \in W'\\
& (\nu x(\bar{x}) \exists y. x = W \land x \land x(\bar{x}))(\bar{x}) \Rightarrow x \in W'\\
& (\nu x(\bar{x}) \exists y. x \models W \land x \land x(\bar{x}))(\bar{x}) \Rightarrow x \in W'\\
& (\nu x(\bar{x}) \exists y. x = W \land x \land x(\bar{x}))(\bar{x}) \Rightarrow x \in W'
\end{array} \]

$\models p_1(\bar{x}) \Rightarrow (\psi' \land (\psi'_1 \lor \psi'_2)) \Rightarrow (\psi' \land ((\psi'_1 \land \psi'_1) \lor (\psi'_2 \land \psi'_2))) \land (\psi'_1 \land \psi'_2)$ and D.5, we obtain $\models (\nu X(\bar{x}).(\nu' \land \psi'_1 \lor \psi'_2))(\bar{x}) \Rightarrow p_1(\bar{x})$.

- **Case Apx'-V**: We have
  - $\emptyset \models 1 \land \psi_1 \\
  - X(\bar{x}) \models p_1; p_2; \nu' \land \psi_1 \\
  - X(\bar{x}) \models p_1; p_2; \nu' \land \psi_2$

By I.H., we obtain

- $\models (\nu X(\bar{x}).(\nu' \land \psi'_1))(\bar{x}) \Rightarrow p_1(\bar{x})$
- $\models (\nu X(\bar{x}).(\nu' \land \psi'_2))(\bar{x}) \Rightarrow p_1(\bar{x})$

By Lemma D.5, we have $\models (\nu X(\bar{x}).(\nu' \land \psi'_1))(\bar{x}) \Rightarrow p_1(\bar{x})$.

- **Case Apx'-V**: We have
  - $\emptyset \models X(\bar{x}) \land X(\bar{x})$
  - $\models p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi'_i$
  - $\models \emptyset \not\models \nu X(\bar{x}) \land X(\bar{f})$
  - $\emptyset \not\models \nu X(\bar{x}) \land X(\bar{f})$
  - $\emptyset \not\models X(\bar{x}) \land p_1; p_2; \psi'_1 \land \psi'_2$
  - $\models \emptyset \not\models i = 1, 2$

By I.H., we obtain $\models (\nu X(\bar{x}).(\nu' \land \psi'_1 \land \psi'_2))(\bar{x}) \Rightarrow p_1(\bar{x})$. By $\models (p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi'_i$ and Lemma D.7, we get $\models (\nu X(\bar{x}).(\nu' \land \nu X(\bar{x}))) \Rightarrow p_1(\bar{x})$.

- **Case Apx'-E**: We have
  - $\emptyset \models X(\bar{x}) \land X(\bar{x})$
  - $\models p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi'_i$
  - $\emptyset \not\models \nu X(\bar{x}) \land X(\bar{x})$
  - $\emptyset \not\models \nu X(\bar{x}) \land X(\bar{x})$
  - $\emptyset \not\models X(\bar{x}) \land p_1; p_2; \psi'_1 \land \psi'_2$
  - $\models \emptyset \not\models i = 1, 2$

By I.H., we obtain $\models (\nu X(\bar{x}).(\nu' \land \psi'_1 \land \psi'_2))(\bar{x}) \Rightarrow p_1(\bar{x})$. By $\models (p_1(\bar{x}) \land \nu' \Rightarrow \exists i.\psi'_i$ and Lemma D.7, we get $\models (\nu X(\bar{x}).(\nu' \land (\nu X(\bar{x}))) \Rightarrow p_1(\bar{x})$. By Lemma D.8 and $\emptyset \not\models \nu X(\bar{x}) \land X(\bar{x})$, we obtain $\models (\nu X(\bar{x}).(\nu' \land \nu X(\bar{x}))) \Rightarrow p_1(\bar{x})$. By Lemma D.8 and $\emptyset \not\models \nu X(\bar{x}) \land X(\bar{x})$, we obtain $\models (\nu X(\bar{x}).(\nu' \land \nu X(\bar{x}))) \Rightarrow p_1(\bar{x})$.
Theorem 5.2 follows from Lemma 5.1.

**Proof of Theorem 5.2.** By induction on the derivation of \( \Gamma \vdash \phi \).

- **Case Fp-Val**: We immediately obtain \( \models \psi \).
- **Case Fp-Lrp**: We obtain
  \[
  \phi = C^\ast[\mu X(\tilde{x}), \psi](\tilde{t})
  \]
  \[
  \models [\tilde{x}, \psi]/X \models \psi'
  \]
  \[
  \models C[\tilde{t}/\tilde{x}][\psi']
  \]
  By I.H., we obtain \( \models C[\tilde{t}/\tilde{x}][\psi'] \). It then follows that \( (\mu X(\tilde{x}), \psi)(\tilde{t}) \models \psi' \). Thus, we get \( \models C^\ast[\mu X(\tilde{x}), \psi](\tilde{t}) \).
- **Case Fp-Grp**\( \ast \): We have
  \[
  \phi = C^\ast[\mu X(\tilde{x}), \psi](\tilde{t})
  \]
  \[
  \models [\tilde{x}, \psi]/X \models \psi'
  \]
  \[
  \models C[\tilde{t}/\tilde{x}][\psi']
  \]
  By I.H., we get \( \models C^\ast[\tilde{t}/\tilde{x}][\psi'] \). Therefore, we have \( \models \psi' \Rightarrow (\nu X(\tilde{x}), \psi)(\tilde{t}) \). We thus get \( \models C^\ast[(\nu X(\tilde{x}), \psi)(\tilde{t})] \).
- **Case Fp-Lrp**\( \ast \): We obtain
  \[
  \phi = C^\ast[\mu X(\tilde{x}), \psi](\tilde{t})
  \]
  \[
  X(\tilde{x}); p_1; p_2; T \Downarrow \text{nff}(\psi)
  \]
  \[
  \models C^\ast[p_1(\tilde{t})]
  \]
  \[
  \models WP(p_2)
  \]
  By I.H., we get \( \models C^\ast[p_1(\tilde{t})] \). By Lemma 5.1, we obtain \( p_1(\tilde{t}) \Rightarrow (\mu X(\tilde{x}), \psi)(\tilde{t}) \). We thus get \( \models C^\ast[(\mu X(\tilde{x}), \psi)(\tilde{t})] \).
- **Case Fp-Grp**\( \ast \): We have
  \[
  X(\tilde{x}); p_1; p_2; T \Downarrow \text{nff}(\psi)
  \]
  \[
  \models C[\tilde{t}/\tilde{x}][\psi]
  \]
  \[
  \models WP(p_2)
  \]
  By I.H., we get \( \models C[\tilde{t}/\tilde{x}][\psi] \). By Lemma 5.1, we obtain \( (\nu X(\tilde{x}), \psi)(\tilde{t}) \Rightarrow \neg p_1(\tilde{t}) \). Therefore, we get \( \models C[\neg p_1(\tilde{t})] \).

\( \square \)

**E. Proof of Theorem 4.1**

**Lemma E.1.** If \( \models [\Gamma \vdash \phi] \), then \( \models [\theta(\rho(\phi))] \) for any predicate substitution \( \rho \) and value substitution \( \theta \) such that \( \text{fpv}(\Gamma) \cup \text{fpv}(\phi) \subseteq \text{dom}(\rho) \) and \( \theta \models [\Gamma] \).

**Lemma E.2 (Soundness of Subtyping).**

- If \( \models [\Gamma \vdash \phi] \), then \( \models [\rho(\phi)] \) for any predicate substitution \( \rho \) with \( \text{fpv}(\Gamma) \cup \text{fpv}(\phi) \subseteq \text{dom}(\rho) \).
- If \( \models [\Gamma \vdash \phi] \), then \( \models [\rho(\phi)] \) for any predicate substitution \( \rho \) with \( \text{fpv}(\Gamma) \cup \text{fpv}(\phi) \subseteq \text{dom}(\rho) \).

**Proof.** By mutual induction on the derivations of \( \Gamma \vdash \phi \).

- **Case S-Inr**: We have
  \[
  \tau_1 = \{ u \mid \phi_1 \}, \tau_2 = \{ u \mid \phi_2 \}
  \]
  \[
  \Gamma \vdash \phi_1 \Rightarrow \phi_2
  \]
  By Theorem 5.2 and Lemma E.1, we get \( \models [\theta(\rho(\phi_1)) \Rightarrow \phi_2(\phi_2)] \).
  Then we get \( [n] \models [n/u] \theta(\rho(\phi_1)) \subseteq [n] \models [n/u] \rho(\phi_2) \).
  By the definition of \( [n] \models [\phi] \), we obtain \( [n] \models [\theta(\rho(\phi_1))] \subseteq [n] \models [\theta(\rho(\phi_2))] \).
  Therefore, we have
  \[
  \models [\theta(\rho(\phi_1))] \subseteq [\theta(\rho(\phi_2))]
  \]
- **Case S-Null**: We have
  \[
  \tau_1 = \{ x \mid \tau_1' \}, \tau_2 = \{ x \mid \tau_2' \}
  \]
  \[
  \Gamma \vdash \tau_1' \Rightarrow \tau_1'
  \]
  By I.H., we get
  \[
  \models \tau_1' \Rightarrow \tau_1'
  \]
  \[
  \models \tau_2' \Rightarrow \tau_1'
  \]
  \[
  \models \tau_2' \Rightarrow \tau_2'
  \]
  By Lemma E.1, we get
  \[
  \models \tau_2' \Rightarrow \tau_2'
  \]
  We then get
  \[
  \models [\theta(\rho(\phi_1))] \subseteq [\theta(\rho(\phi_2))]
  \]
  and \( \models [\theta(\rho(\phi_1))] \subseteq [\theta(\rho(\phi_2))] \).

**Lemma E.3** (Effect Composition). For any \( \Phi_1 \) and \( \Phi_2 \), we have:

- For any \( \phi_1 \) and \( \phi_2 \), if \( \models [\Phi_1(\phi_1)] \) and \( [\Phi_2(\phi_2)] \), then \( \models [\Phi_1 \circ \Phi_2(\phi_1) \circ \Phi_2(\phi_2)] \).
- For any \( \phi_1 \) and \( \phi_2 \), if \( \models [\Phi_1(\phi_1)] \) and \( \models [\Phi_2(\phi_2)] \), then \( \models [\Phi_1 \circ \Phi_2(\phi_1) \circ \Phi_2(\phi_2)] \).
- For any \( \pi_1 \), if \( \models [\Phi_1(\pi_1)] \), then \( \models [\Phi_1 \circ \Phi_2(\pi_1)] \).

**Lemma E.4.** Let \( \lambda X.\overline{X.}\psi \) with \( \overline{X.} \) may occur only positively in \( \psi \). We have \( \mu X(\overline{X.}) \models [\psi] \subseteq \bigcup_{i=0}^{\infty} F_i(\lambda \overline{X.}) \).

**Proof.** \( \models [F(\bigcup_{i=0}^{\infty} F_i(\lambda \overline{X.}))] \subseteq \bigcup_{i=0}^{\infty} F_i(\lambda \overline{X.}) \) is obtained by the continuity of \( F \), which is proved by induction on the structure of \( \psi \). \( \square \)

**Lemma E.5.** Let \( \lambda X.\overline{X.}\psi \) with \( \overline{X.} \) may occur only positively in \( \psi \). We have \( \bigcup_{i=0}^{\infty} F_i(\lambda \overline{X.}) \subseteq \nu X(\overline{X.}) \).
Proof. \( \vdash \bigwedge_{i=0}^{\infty} F_i((\phi, \tau)) \subseteq F_i((\phi, \tau)) \) is obtained by the cocontinuity of \( F \), which we get by induction on the structure of \( \psi \).

\( \square \)

Lemma E.6. Let \( \tau_f' = (\chi, \tau) \rightarrow (\tau \& (\lambda x \in \Sigma^* \text{mu}(\chi, x), \lambda x \in \Sigma^* \text{mu}(\chi, x))) \) and \( \rho \) be a predicate substitution with \( dom(\rho) = f_{\text{sub}}(\tau_f') \setminus \{X_\mu, X_v\} \). Suppose that \( e \in \rho(f : \rho'(\tau_f'), \chi, \tau) + \rho(\tau \& \rho'(\Phi)) \) for any \( \rho'(\tau_f') = \{X_\mu, X_v\} \). We then have \( \rho(e, \chi, e) \in \{f(\tau_f') & F_0(\tau_f') \} \), where

- \( q_\mu = \text{mu}(\chi, x).\Phi(x) \)
- \( q_v = \text{nu}(\chi, x).[q_\mu/q_\mu].\Phi(x) \)
- \( \tau_f = (\chi, \tau) \rightarrow (\tau \& (\lambda x \in \Sigma^* q_\mu(x), \lambda x \in \Sigma^* q_v(x))) \)

Proof. By Theorem 5.2 and Lemma E.1, for any predicate substitution \( \rho' \) with \( dom(\rho') = \{X_\mu, X_v\} \) and value substitution \( \theta \) with \( \theta \models \rho(f : \rho'(\tau_f'), \chi, \tau) \), we obtain

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

By (6), it follows that \( w \in [\theta(\rho(\tau))] \Rightarrow \exists i. \models \theta(\rho(p_i'(\omega, \tau))) \). By Lemma E.4, we get

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(7)

Therefore, we get

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

We next prove

\[
\forall \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(8)

We first show

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

By (6), it follows that \( w \in [\theta(\rho(\tau))] \Rightarrow \exists i. \models \theta(\rho(p_i'(\omega, \tau))) \). By Lemma E.4, we get

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(7)

Here, \( e^\tau \) is a closed expression such that \( e^\tau \uparrow \& \Re \). By mathematical induction, we will prove that for all \( i, w, \omega_0 \),

\[
(e^\tau_e \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(8)

- We show that (8) holds for \( i = 0 \). Because \( e^\tau \) is closed, we get \( \theta(\rho'(\Phi)) = e^\tau = e_{\text{mu}}^{\rho}(\chi, x).\downarrow \). By (6), we get

\[
\forall \omega_0. (\theta(e^\tau_{\text{mu}} \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

- We show that (8) holds for \( i = 0 \). Because \( e^\tau \) is closed, we get \( \theta(\rho'(\Phi)) = e^\tau = e_{\text{mu}}^{\rho}(\chi, x).\downarrow \). By (6), we get

\[
\forall \omega_0. (\theta(e^\tau_{\text{mu}} \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

We next prove

\[
\forall \omega_0. (\theta(e^\tau_{\text{mu}} \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(7)

Here, \( e^\tau \) is a closed expression such that \( e^\tau \uparrow \& \Re \). By mathematical induction, we will prove that for all \( i, w, \omega_0 \),

\[
(e^\tau_e \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(8)

- We show that (8) holds for \( i = 0 \). Because \( e^\tau \) is closed, we get \( \theta(\rho'(\Phi)) = e^\tau = e_{\text{mu}}^{\rho}(\chi, x).\downarrow \). By (6), we get

\[
\forall \omega_0. (\theta(e^\tau_{\text{mu}} \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

- We show that (8) holds for \( i = 0 \). Because \( e^\tau \) is closed, we get \( \theta(\rho'(\Phi)) = e^\tau = e_{\text{mu}}^{\rho}(\chi, x).\downarrow \). By (6), we get

\[
\forall \omega_0. (\theta(e^\tau_{\text{mu}} \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(6)

By (6) with \( \rho' = \{X_\mu \mapsto \rho'_i, X_v \mapsto \rho'_i(\chi, x).\downarrow \} \), we obtain

\[
\forall \omega, \omega_0. (\theta(e) \downarrow \& \theta(\Phi)) \Rightarrow w \in [\theta(\rho(\tau))] \Rightarrow \theta(\rho(\rho'(\Phi))(\omega))
\]

(7)
Therefore, by RT-App and RN-App, we get
\[\forall w, \alpha, (\text{rec}(f, \bar{x}, \alpha), \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(w \land \alpha) \Rightarrow w \in \{0(\rho(\tau))\} \land \theta(\rho(p_{i+1}(x, x)))\]
\[\forall \pi, (\text{rec}(f, \bar{x}, \alpha), \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x))) \Rightarrow \rho(\rho(p_{i+1}(x, x))) \land \theta(p_{i+1}(x, x)))\]
We then get \(\forall w, \alpha, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(w \land \alpha) \Rightarrow w \in \{0(\rho(\tau))\} \land \theta(\rho(p_{i+1}(x, x)))\)
We get \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x))) \Rightarrow \rho(\rho(p_{i+1}(x, x))) \land \theta(p_{i+1}(x, x)))\).

By (8), we have
\[\forall \pi, \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x))) \Rightarrow \rho(\rho(p_{i+1}(x, x))) \land \theta(p_{i+1}(x, x)))\]
We then get \(\forall \pi, \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x))) \Rightarrow \rho(\rho(p_{i+1}(x, x))) \land \theta(p_{i+1}(x, x)))\).
Suppose that \(\text{rec}(f, \bar{x}, e, \theta(\bar{x})) \in \theta(\bar{x}) \iff \theta(\rho(q_x(X_p, p_{i+1}(x, x))))\).
By Lemma 6, \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x))) \Rightarrow \rho(\rho(p_{i+1}(x, x))) \land \theta(p_{i+1}(x, x)))\).

By the definition of \([\bar{x} \rightarrow \sigma]\), we obtain \(\forall \pi, \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\rho(p_{i+1}(x, x)))\).

Proof of Theorem 4.1. By induction on the derivation of \(\Gamma \vdash e : \sigma\).

Let \(\theta\) be a value substitution such that \(\theta \models \Gamma\).

**Case T-Const:** We have
- \(\sigma = \epsilon = n\)
- \(\sigma = \{x \mid x = n\} \in \Phi_{val}\)

There is no \(\pi\) such that \(n \not\in \pi \land \pi\) because no rule for NonTerminating Run applies to \(n\). Then, we get \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Also, RT-Val is the only rule for Terminating Run that applies to \(n\). Therefore, if \(n \not\in \alpha \land \alpha\), then \(n \not\in \alpha \land \alpha\). Then we get \(\forall \pi, \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

By Lemma 6, \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Thus, we get \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

**Case T-VFun:** We have
- \(e = x \lor \sigma = \{x \mid x = n\} \in \Phi_{val}\)
- \(\Gamma \vdash e_1 : (\tau' \land \Phi) \lor \tau' \land \Phi_{val}\)
- \(\Gamma \vdash e_2 : (\tau' \land \Phi) \lor \tau' \land \Phi_{val}\)

By LH, Theorem 5.2, and Lemma 6.3, we get \(\forall \pi, \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Thus, we get \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Thus, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Thus, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).

Therefore, we have \(\forall \pi, (\alpha, \bar{v}_{k+1}^e, f) \in \theta(\bar{x}) \iff \theta(\mu(\bar{p}_{i+1}(x, x)))\).
\[\begin{align*}
\text{Case T-If:} & \quad \text{We have} \\
& - \varepsilon = \text{ifz } v \text{ then } e_1 \text{ else } e_2 \\
& - \sigma = (r \& \Phi) \\
& - \Gamma \vdash v \in \left[\Gamma \vdash (\text{int } \& \Phi_{val})\right] \\
& - \text{By IH, we obtain} \\
& v_1 \in \left[\Gamma \vdash (\text{int } \& \Phi_{val})\right] \text{ and } v_2 \in \left[\Gamma \vdash (\text{int } \& \Phi_{val})\right]. \text{Because there is no rule for Non Terminating Run that applies to } \theta(v_1 \text{ op } v_2), \text{ there is no } \pi \text{ such that } \theta(v_1 \text{ op } v_2) \parallel \& \pi. \text{ We then get } V \forall \pi. \left(\theta(v_1 \text{ op } v_2) \parallel \& \pi \right) \Rightarrow \theta(\rho(\Phi'_{val})(\pi)). \text{ Also,} \\
& \text{RT-Or is the only rule for Terminating Run that applies to } \theta(v_1 \text{ op } v_2). \text{ Therefore, if } \theta(v_1 \text{ op } v_2) \parallel w \& \omega, \text{ then } w = \left[\text{op}\right](\theta(v_1), \theta(v_2)) \text{ and } \omega = e. \text{ We have } V \forall \omega. w \& \omega(v_1 \text{ op } v_2) \parallel w \& \omega \Rightarrow w = \rho(\left[\text{op}\right](\theta(v_1), \theta(v_2))) \wedge \theta(\rho(\Phi'_{val})(\omega)). \text{ We then get } \theta(v_1 \text{ op } v_2) \in \left[\theta(\rho[x | x = v_1 \text{ op } v_2] \& \Phi_{val})\right]. \text{ It follows that} \\
& v_1 \text{ op } v_2 \in \left[\rho(\Gamma) \vdash \rho((\{x | x = 0\} \& (\lambda x \in \Sigma^* . x \in a \& \lambda x \in \Sigma^{\omega . \omega} . \_)))).\right]
\end{align*}\]

**Case T-Sun:** We have

- \(\sigma = \sigma_2\)
- \(\Gamma \vdash e : \sigma_1\)
- \(\Gamma \vdash \sigma_1 \triangleleft \sigma_2\)

By IH, we obtain \(e \in \left[\rho(\Gamma) \vdash \rho(\sigma_1)\right].\) By the definition, we get \(V \forall \theta \in \text{styp}(\Gamma), (\theta \vdash \rho(\Gamma)) \Rightarrow \theta(e) \in \left[\theta(\rho(\sigma_1))\right].\) By Lemma \(E.2,\) we have \(\forall \theta \in \text{styp}(\Gamma), (\theta \vdash \rho(\Gamma)) \Rightarrow \theta(e) \in \left[\theta(\rho(\sigma_2))\right].\) Therefore, we get \(V \forall \theta \in \text{styp}(\Gamma), (\theta \vdash \rho(\Gamma)) \Rightarrow \theta(e) \in \left[\theta(\rho(\sigma_2))\right].\) Thus we have \(e \in \left[\rho(\Gamma) \vdash \rho(\sigma_2)\right].\)

\[\square\]

### F Example Typing Derivation

Figure 16 shows an example typing derivation.

### G Example Fixpoint Approximation

Figures 17-19 show example under-approximations of least fixpoints. Figures 20-21 show example over-approximations of greatest fixpoints.

### H Semantics of First-Order Fixpoint Logic

We define the predicate sorts \(T\) and the partially ordered set \((D_S, \preceq_S)\) and \((\mathcal{D}_T, \subseteq_T)\) inductively by:

\[\text{(predicate sorts) } \mathcal{T} := \bullet \Rightarrow [\overline{S} \rightarrow \bullet]\]

\[\begin{align*}
D_S & \triangleq \{T, \bot\} \\
D_T & \triangleq \mathbb{Z} \\
D_X & \triangleq \{\pi, \omega\} \quad (\pi \in \mathbb{Z}, \omega \in \Sigma^*) \\
D_Y & \triangleq \{\pi, \omega\} \quad (\pi \in \Sigma^*) \\
D_{\overline{f}} & \triangleq \{f \in D_{\overline{f}}, \forall d_1, d_2 \in D_{\overline{S}}, d_1 \preceq_S d_2 \Rightarrow f(d_1) \preceq f(d_2)\} \\
\overline{f} & \triangleq \{(f, g) | \forall d \in D_{\overline{S}} . f(d) \preceq_S g(d)\}
\end{align*}\]

and, we define conjunction and disjunction of \(f, g \in D_{\overline{f}}\) as follows:

\[\begin{align*}
f \land \overline{f} & \triangleq \lambda d \in D_{\overline{f}} . f(d) \land g(d) \\
f \lor \overline{f} & \triangleq \lambda d \in D_{\overline{f}} . f(d) \lor g(d)
\end{align*}\]

Note that \((D_{\overline{f}}, \subseteq_{\overline{f}})\) forms a complete lattice. (See Theorem H.1). The least and greatest elements of \((D_{\overline{f}}, \subseteq_{\overline{f}})\) are \(\overline{\lambda} X, \bot\) respectively.

We now define the normal semantics of the first-order fixpoint logic as Fig 22. \text{sort}(A) and \text{sort}(f) are the sorts sequence of arguments of \(A\) and \(f\) respectively.

Here, the least/greatest fixpoint operators \(\text{lfp}_{\overline{f}}\) and \(\text{gfp}_{\overline{f}}\) are defined by:

\[\begin{align*}
\text{lfp}_{\overline{f}}(F) & \triangleq \bigwedge_{\overline{f}} \{X \in D_{\overline{f}} | F(X) \subseteq_{\overline{f}} X\} \\
\text{gfp}_{\overline{f}}(F) & \triangleq \bigvee_{\overline{f}} \{X \in D_{\overline{f}} | X \subseteq_{\overline{f}} F(X)\}
\end{align*}\]

\[\bigwedge_{\overline{f}}\text{ and } \bigvee_{\overline{f}}\text{ denote respectively the greatest lower bound and the least upper bound with respect to } \subseteq_{\overline{f}}\]

**Theorem H.1.** \((D_{\overline{f}}, \subseteq_{\overline{f}})\) forms a complete lattice.
\[ (\lambda x. x = a, x \leq x) \]

Figure 16. Typing Derivation for \( \text{rec}(f, n, if z n \text{ then } 1 \text{ else } (ev[a]; let n' = n - 1 in f n')) (\tau_f) \)

\[
\begin{align*}
\Gamma_e &\vdash \tau_f' n : \text{int}, n = 0 & \Phi_n = (\lambda x. x = a, x \leq x, x) \\
\Gamma_e &\vdash \tau_f' n : \text{int}, n \neq 0 & \sigma_{\text{int}} = (\text{int} \& \Phi_n) \\
\tau_f' &\equiv (n : \text{int}) \rightarrow (\{x \mid x = 1\} \& (\lambda x. X_\mu(n, x), \lambda x. X_\nu(n, x))) \\
\Phi' &\equiv (\lambda x. X_\mu(n', x), \lambda x. X_\nu(n', x)) \\
\phi &\equiv \left( \begin{array}{c}
\lambda x. n = 0 \land x = e \lor n \neq 0 \land \exists y. x = a \cdot y \land X_\mu(n - 1, y) \\
\lambda x. n \neq 0 \land \exists y. x = a \cdot y \land X_\nu(n - 1, y)
\end{array} \right) \\
q_\nu &\equiv \mu X_\nu(n, x). \Phi' (x) \\
q_\nu &\equiv \nu X_\nu(n, x). [q_\mu / X_\mu] \Phi' (x) \\
\tau_f &\equiv (n : \text{int}) \rightarrow (\{x \mid x = 1\} \& (\lambda x. q_\mu(n, x), \lambda x. q_\nu(n, x)))
\end{align*}
\]

Figure 17. The least fixpoint \( \mu X(n). (n = 0 \lor (n \neq 0) \land X(n - 1)) \) is under-approximated by \( \lambda n. n \geq 0 \)

\[
\begin{align*}
\vdash (p_1(n) \land \psi_1) &\Rightarrow n = 0 & \vdash (p_1(n) \land \psi_2) \Rightarrow (p_1(n - 1) \land p_2(n, n - 1)) \\
X(n); p_1; p_2; \psi_1 &\downarrow n = 0 & X(n); p_1; p_2; \psi_2 &\downarrow n \neq 0 \land X(n - 1) \\
X(n); p_1; p_2; \psi_1 &\downarrow n \neq 0 \land X(n - 1) & X(n); p_1; p_2; \psi_2 &\downarrow n \neq 0 \land X(n - 1) \\
X(n); p_1; p_2; \tau &\downarrow n = 0 \lor (n \neq 0) \land X(n - 1) & X(n); p_1; p_2; \tau &\downarrow n = 0 \lor (n \neq 0) \land X(n - 1)
\end{align*}
\]

Here, \( \psi_1 \equiv (n = 0) \), \( \psi_2 \equiv (n \neq 0) \), \( p_1 \equiv \lambda n. n \geq 0 \), \( p_2 \equiv \lambda n_1, n_2, n_1 > n_2 \geq 0 \)

Figure 18. The least fixpoint \( \mu X(n, x). n = 0 \lor n \neq 0 \land \exists y. x = a \cdot y \land X(n - 1, y) \) is under-approximated by \( \lambda (n, x). n \geq 0 \land x \in \mathbb{R}^n \)

**Proof.** For any subset of \( D_{\tau_f} \), \( \{p_0, p_1, \cdots, p_i\} \), these hold.

So, this has the least element \( \lambda \tilde{d} \in D_{\tau_f} \) and the greatest element \( \lambda \tilde{d} \in D_{\tau_f} \).

\[
\begin{align*}
\langle \lambda \tilde{d} \in D_{\tau_f} \rangle \subseteq &\vdash_{\tau_f} p_k \\
p_k \subseteq &\vdash_{\tau_f} (\lambda \tilde{d} \in D_{\tau_f} \cup \bigcup_{j=0}^{i} p_j(\tilde{d}))
\end{align*}
\]

\[ \square \]
\[
\vDash (p_1(n) \land n = 0 \land x = \epsilon) \Rightarrow (n = 0 \land x = \epsilon)
\]

\[
\frac{\vDash (p_1(n), x) \Rightarrow (n = 0 \land x = \epsilon) \land n \neq 0 \land \text{hd}(x) = a \lor \text{hd}(x) = b}{\vDash (p_1(n), x) \Rightarrow (n = 0 \land x = \epsilon) \lor n \neq 0 \land \text{hd}(x) = a \lor \text{hd}(x) = b}
\]

\[
\vDash (p_1(n) \land n \neq 0) \Rightarrow n \neq 0
\]

Here, \(\vDash X(n, x) ; p_1 ; p_2; \top \equiv (n \neq 0 \land \exists y, x = a \cdot y \land X(n, y)) \lor (\exists y, x = b \cdot y \land X(n, y)) \Rightarrow (p_1(n) \land n \neq 0 \land \text{hd}(x) = a) \Rightarrow \exists x'_1, tl(x) = x'_1
\]

**Figure 19.** The least fixpoint \(\mu X(n, x) (n = 0 \land x = \epsilon) \lor (n \neq 0 \land \exists y, x = a \cdot y \land X(n, y)) \lor (\exists y, x = b \cdot y \land X(n, y))\) is under-approximated by \(A \cdots X(n, x) ; p_1 ; p_2; \text{hd}(x) = b \lor \exists y, x = a \cdot y \land X(n, y) \Rightarrow (p_1(n) \land \text{hd}(x) = b) \Rightarrow \exists x'_2, tl(x) = x'_2\)

\[
\vDash (p_1(n) \land \psi_2) \Rightarrow x = b \cdot x'_2 \quad \vDash (p_1(n) \land \psi_2) \Rightarrow (p_1(n, x'_2) \land p_2(n, x, x'_2))
\]

Here, \(\vDash X(n, x) ; p_1 ; p_2; \psi_2 \iff x = b \cdot x'_2 \land X(n, x'_2) \Rightarrow (p_1(n) \land \psi_2)
\]

**Figure 20.** The greatest fixpoint \(\nu X(n) ; n \neq 0 \land X(n - 1)\) is over-approximated by \(\lambda n \cdot \neg (n \neq 0) \equiv \lambda n, n < 0\)

\[
\vDash (p_1(n) \land x \neq a \cdot x') \Rightarrow \neg(x = a \cdot x') \quad \vDash (p_1(n) \land x = a \cdot x') \Rightarrow (p_1(x') \land p_2(x, x'))
\]

Here, \(\vDash X(x) ; p_1 ; p_2; x \neq a \cdot x' \Rightarrow X(x') \Rightarrow (x \neq a \cdot x' \lor x = a \cdot x')
\]

\[
\frac{\vDash (p_1(x) \land x = a \cdot x') \Rightarrow \neg(x \neq a \cdot x') \lor x = a \cdot x')}{\vDash (p_1(x) \land x = a \cdot x') \Rightarrow \neg(x \neq a \cdot x') \lor x = a \cdot x')}
\]

\[
\frac{\vDash (p_1(x) \land x = a \cdot x') \Rightarrow \neg(x \neq a \cdot x') \lor x = a \cdot x')}{\vDash (p_1(x) \land x = a \cdot x') \Rightarrow \neg(x \neq a \cdot x') \lor x = a \cdot x')}
\]

Here, \(p_1 \equiv \lambda x, x \not\in a^\omega, p_2 \equiv \lambda x_1, x_2, (\text{first position of non-} a \text{ in } x_1) > (\text{first position of non-} a \text{ in } x_2) \geq 0\)

**Figure 21.** The greatest fixpoint \(\nu X(x) ; x = a \cdot y \land X(y)\) is over-approximated by \(\lambda x, \neg(x \not\in a^\omega) \equiv \lambda x, x \in a^\omega\)
\[ \begin{align*}
\theta \models \top \\
\theta \not\models \bot \\
\theta \models \neg \phi \iff \theta \not\models \phi \\
\theta \models \phi_1 \land \phi_2 \iff \theta \models \phi_1 \text{ and } \theta \models \phi_2 \\
\theta \models \phi_1 \lor \phi_2 \iff \theta \models \phi_1 \text{ or } \theta \models \phi_2 \\
\theta \models \forall x : S . \phi \text{ iff for all } d \in D_S, [x \mapsto d] \theta \models \phi \\
\theta \models \exists x : S . \phi \text{ iff for some } d \in D_S, [x \mapsto d] \theta \models \phi \\
\theta \models A(\bar{t}) \iff [A]_\theta \in D_{\text{sort}(A)} \text{ and } \models [A](\bar{f}_\theta) \\
\theta \models X(\bar{t} : S) \iff [\Pi]_\theta \in D_S \text{ and for all predicates } p, \models p(\bar{t}) \\
\theta \models (\mu X(\bar{x} : S) . \phi)(\bar{t}) \iff \theta \models \text{lfp } e \cdot \bar{t} \Rightarrow (\lambda X . \lambda \bar{x} . \phi)(\bar{t}) \\
\theta \models (\nu X(\bar{x} : S) . \phi)(\bar{t}) \iff \theta \models \text{gfp } e \cdot \bar{t} \Rightarrow (\lambda X . \lambda \bar{x} . \phi)(\bar{t}) \\
[x : S]_\theta \triangleq \theta(x) \in D_S \\
[f(\bar{t})]_\theta \triangleq [f](\bar{f}_\theta) \in D_S \quad \text{(where } \bar{t} \in D_{\text{sort}(f)})
\end{align*} \]

**Figure 22.** Semantics of Fixpoint logic formula