Higher-Order Program Verification via HFL Model Checking

Naoki Kobayashi(✉), Takeshi Tsukada, and Keiichi Watanabe

The University of Tokyo, Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Abstract. There are two kinds of higher-order extensions of model checking: HORS model checking and HFL model checking. Whilst the former has been applied to automated verification of higher-order functional programs, applications of the latter have not been well studied. In the present paper, we show that various verification problems for functional programs, including may/must-reachability, trace properties, and linear-time temporal properties (and their negations), can be naturally reduced to (extended) HFL model checking. The reductions yield a sound and complete logical characterization of those program properties. Compared with the previous approaches based on HORS model checking, our approach provides a more uniform, streamlined method for higher-order program verification.

1 Introduction

There are two kinds of higher-order extensions of model checking in the literature: HORS model checking [16,32] and HFL model checking [42]. The former is concerned about whether the tree generated by a given higher-order tree grammar called a higher-order recursion scheme (HORS) satisfies the property expressed by a given modal μ-calculus formula (or a tree automaton), and the latter is concerned about whether a given finite state system satisfies the property expressed by a given formula of higher-order modal fixpoint logic (HFL), a higher-order extension of the modal μ-calculus. Whilst HORS model checking has been applied to automated verification of higher-order functional programs [17,18,22,26,33,41,43], there have been few studies on applications of HFL model checking to program/system verification. Despite that HFL has been introduced more than 10 years ago, we are only aware of applications to assume-guarantee reasoning [42] and process equivalence checking [28].

In the present paper, we show that various verification problems for higher-order functional programs can actually be reduced to (extended) HFL model checking in a rather natural manner. We briefly explain the idea of our reduction below.[1] We translate a program to an HFL formula that says "the program has a valid behavior" (where the *validity* of a behavior depends on each verification

[1] In this section, we use only a fragment of HFL that can be expressed in the modal μ-calculus. Some familiarity with the modal μ-calculus [25] would help.

problem). Thus, a program is actually mapped to a *property*, and a program property is mapped to a system to be verified; this has been partially inspired by the recent work of Kobayashi et al. [19], where HORS model checking problems have been translated to HFL model checking problems by switching the roles of models and properties.

For example, consider a simple program fragment $\texttt{read}(x); \texttt{close}(x)$ that reads and then closes a file (pointer) x. The transition system in Fig. 1 shows a valid access protocol to read-only files. Then, the property that a read operation is allowed in the current state can be expressed by a formula of the form $\langle\texttt{read}\rangle\varphi$, which says that the current state has a $\texttt{read}$-transition, after which φ is satisfied. Thus, the program $\texttt{read}(x); \texttt{close}(x)$ being valid is expressed as $\langle\texttt{read}\rangle\langle\texttt{close}\rangle\textbf{true}$,[2] which is indeed satisfied by the initial state q_0 of the transition system in Fig. 1. Here, we have just replaced the operations $\texttt{read}$ and $\texttt{close}$ of the program with the corresponding modal operators $\langle\texttt{read}\rangle$ and $\langle\texttt{close}\rangle$. We can also naturally deal with branches and recursions. For example, consider the program $\texttt{close}(x)\Box(\texttt{read}(x); \texttt{close}(x))$, where $e_1\Box e_2$ represents a non-deterministic choice between e_1 and e_2. Then the property that the program always accesses x in a valid manner can be expressed by $(\langle\texttt{close}\rangle\textbf{true}) \wedge (\langle\texttt{read}\rangle\langle\texttt{close}\rangle\textbf{true})$. Note that we have just replaced the non-deterministic branch with the logical conjunction, as we wish here to require that the program's behavior is valid in *both* branches. We can also deal with conditional branches if HFL is extended with predicates; $\textbf{if}\ b\ \textbf{then}\ \texttt{close}(x)\ \textbf{else}\ (\texttt{read}(x); \texttt{close}(x))$ can be translated to $(b \Rightarrow \langle\texttt{close}\rangle\textbf{true}) \wedge (\neg b \Rightarrow \langle\texttt{read}\rangle\langle\texttt{close}\rangle\textbf{true})$. Let us also consider the recursive function f defined by:

$$f\,x = \texttt{close}(x)\Box(\texttt{read}(x); \texttt{read}(x); f\,x),$$

Then, the program $f\,x$ being valid can be represented by using a (greatest) fixpoint formula:

$$\nu F.(\langle\texttt{close}\rangle\textbf{true}) \wedge (\langle\texttt{read}\rangle\langle\texttt{read}\rangle F).$$

If the state q_0 satisfies this formula (which is indeed the case), then we know that all the file accesses made by $f\,x$ are valid. So far, we have used only the modal μ-calculus formulas. If we wish to express the validity of higher-order programs, we need HFL formulas; such examples are given later.

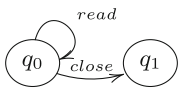


Fig. 1. File access protocol

[2] Here, for the sake of simplicity, we assume that we are interested in the usage of the single file pointer x, so that the name x can be ignored in HFL formulas; usage of multiple files can be tracked by using the technique of [17].

We generalize the above idea and formalize reductions from various classes of verification problems for simply-typed higher-order functional programs with recursion, integers and non-determinism – including verification of may/must-reachability, trace properties, and linear-time temporal properties (and their negations) – to (extended) HFL model checking where HFL is extended with integer predicates, and prove soundness and completeness of the reductions. Extended HFL model checking problems obtained by the reductions are (necessarily) undecidable in general, but for finite-data programs (i.e., programs that consist of only functions and data from finite data domains such as Booleans), the reductions yield *pure* HFL model checking problems, which are decidable [42].

Our reductions provide sound and complete logical characterizations of a wide range of program properties mentioned above. Nice properties of the logical characterizations include: (i) (like verification conditions for Hoare triples,) once the logical characterization is obtained as an HFL formula, purely logical reasoning can be used to prove or disprove it (without further referring to the program semantics); for that purpose, one may use theorem provers with various degrees of automation, ranging from interactive ones like Coq, semi-automated ones requiring some annotations, to fully automated ones (though the latter two are yet to be implemented), (ii) (unlike the standard verification condition generation for Hoare triples using invariant annotations) the logical characterization can *automatically* be computed, without any annotations,[3] (iii) standard logical reasoning can be applied based on the semantics of formulas; for example, co-induction and induction can be used for proving ν- and μ-formulas respectively, and (iv) thanks to the completeness, the set of program properties characterizable by HFL formula is closed under negations; for example, from a formula characterizing may-reachability, one can obtain a formula characterizing non-reachability by just taking the De Morgan dual.

Compared with previous approaches based on HORS model checking [18, 22,26,33,37], our approach based on (extended) HFL model checking provides more uniform, streamlined methods for higher-order program verification. HORS model checking provides sound and complete verification methods for *finite-data* programs [17,18], but for infinite-data programs, other techniques such as predicate abstraction [22] and program transformation [27,31] had to be combined to obtain sound (but incomplete) reductions to HORS model checking. Furthermore, the techniques were different for each of program properties, such as reachability [22], termination [27], non-termination [26], fair termination [31], and fair non-termination [43]. In contrast, our reductions are sound and complete even for infinite-data programs. Although the obtained HFL model checking problems are undecidable in general, the reductions allow us to treat various program properties uniformly; all the verifications are boiled down to the issue of how to prove μ- and ν-formulas (and as remarked above, we can use induction and co-induction to deal with them). Technically, our reduction to HFL model

[3] This does not mean that invariant discovery is unnecessary; invariant discovery is just postponed to the later phase of discharging verification conditions, so that it can be uniformly performed among various verification problems.

checking may actually be considered an extension of HORS model checking in the following sense. HORS model checking algorithms [21,32] usually consist of two phases, one for computing a kind of higher-order "procedure summaries" in the form of variable profiles [32] or intersection types [21], and the other for nested least/greatest fixpoint computations. Our reduction from program verification to extended HFL model checking (the reduction given in Sect. 7, in particular) can be regarded as an extension of the first phase to deal with infinite data domains, where the problem for the second phase is expressed in the form of extended HFL model checking: see [23] for more details.

The rest of this paper is structured as follows. Section 2 introduces HFL extended with integer predicates and defines the HFL model checking problem. Section 3 informally demonstrates some examples of reductions from program verification problems to HFL model checking. Section 4 introduces a functional language used to formally discuss the reductions in later sections. Sections 5, 6, and 7 consider may/must-reachability, trace properties, and temporal properties respectively, and present (sound and complete) reductions from verification of those properties to HFL model checking. Section 8 discusses related work, and Sect. 9 concludes the paper. Proofs are found in an extended version [23].

2 (Extended) HFL

In this section, we introduce an extension of higher-order modal fixpoint logic (HFL) [42] with integer predicates (which we call HFL$_{\mathbf{Z}}$; we often drop the subscript and write HFL, as in Sect. 1), and define the HFL$_{\mathbf{Z}}$ model checking problem. The set of integers can actually be replaced by another infinite set X of data (like the set of natural numbers or the set of finite trees) to yield HFL$_X$.

2.1 Syntax

For a map f, we write $dom(f)$ and $codom(f)$ for the domain and codomain of f respectively. We write $\mathbf{Z}$ for the set of integers, ranged over by the metavariable n below. We assume a set **Pred** of primitive predicates on integers, ranged over by p. We write $\mathtt{arity}(p)$ for the arity of p. We assume that **Pred** contains standard integer predicates such as $=$ and $<$, and also assume that, for each predicate $p \in \mathbf{Pred}$, there also exists a predicate $\neg p \in \mathbf{Pred}$ such that, for any integers $n_1, \ldots, n_k$, $p(n_1, \ldots, n_k)$ holds if and only if $\neg p(n_1, \ldots, n_k)$ does not hold; thus, $\neg p(n_1, \ldots, n_k)$ should be parsed as $(\neg p)(n_1, \ldots, n_k)$, but can semantically be interpreted as $\neg(p(n_1, \ldots, n_k))$.

The syntax of *HFL*$_{\mathbf{Z}}$ *formulas* is given by:

$$\begin{array}{rl}
\varphi \text{ (formulas)} ::= & n \mid \varphi_1 \,\mathtt{op}\, \varphi_2 \mid \mathbf{true} \mid \mathbf{false} \mid p(\varphi_1, \ldots, \varphi_k) \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\
& \mid X \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X^{\tau}.\varphi \mid \nu X^{\tau}.\varphi \mid \lambda X : \sigma.\varphi \mid \varphi_1\, \varphi_2 \\
\tau \text{ (types)} ::= & \bullet \mid \sigma \to \tau \qquad \sigma \text{ (extended types)} ::= \tau \mid \mathtt{int}
\end{array}$$

Here, $\mathtt{op}$ ranges over a set of binary operations on integers, such as $+$, and X ranges over a denumerable set of variables. We have extended the original HFL [42] with integer expressions (n and $\varphi_1 \,\mathtt{op}\, \varphi_2$), and atomic formulas

$p(\varphi_1, \ldots, \varphi_k)$ on integers (here, the arguments of integer operations or predicates will be restricted to integer expressions by the type system introduced below). Following [19], we have omitted negations, as any formula can be transformed to an equivalent negation-free formula [30].

We explain the meaning of each formula informally; the formal semantics is given in Sect. 2.2. Like modal μ-calculus [10,25], each formula expresses a property of a labeled transition system. The first line of the syntax of formulas consists of the standard constructs of predicate logics. On the second line, as in the standard modal μ-calculus, $\langle a\rangle\varphi$ means that there exists an a-labeled transition to a state that satisfies φ. The formula $[a]\varphi$ means that after any a-labeled transition, φ is satisfied. The formulas $\mu X^\tau.\varphi$ and $\nu X^\tau.\varphi$ represent the least and greatest fixpoints respectively (the least and greatest X that $X = \varphi$) respectively; unlike the modal μ-calculus, X may range over not only propositional variables but also higher-order predicate variables (of type τ). The λ-abstractions $\lambda X\!:\!\sigma.\varphi$ and applications $\varphi_1\ \varphi_2$ are used to manipulate higher-order predicates. We often omit type annotations in $\mu X^\tau.\varphi$, $\nu X^\tau.\varphi$ and $\lambda X : \sigma.\varphi$, and just write $\mu X.\varphi$, $\nu X.\varphi$ and $\lambda X.\varphi$.

Example 1. Consider $\varphi_{\mathtt{ab}}\,\varphi$ where $\varphi_{\mathtt{ab}} = \mu X^{\bullet\to\bullet}.\lambda Y : \bullet.Y \vee \langle\mathtt{a}\rangle(X(\langle\mathtt{b}\rangle Y))$. We can expand the formula as follows:

$$\begin{aligned}\varphi_{\mathtt{ab}}\,\varphi &= (\lambda Y.\bullet.Y \vee \langle\mathtt{a}\rangle(\varphi_{\mathtt{ab}}(\langle\mathtt{b}\rangle Y)))\varphi = \varphi \vee \langle\mathtt{a}\rangle(\varphi_{\mathtt{ab}}(\langle\mathtt{b}\rangle\varphi))\\ &= \varphi \vee \langle\mathtt{a}\rangle(\langle\mathtt{b}\rangle\varphi \vee \langle\mathtt{a}\rangle(\varphi_{\mathtt{ab}}(\langle\mathtt{b}\rangle\langle\mathtt{b}\rangle\varphi))) = \cdots,\end{aligned}$$

and obtain $\varphi \vee (\langle\mathtt{a}\rangle\langle\mathtt{b}\rangle\varphi) \vee (\langle\mathtt{a}\rangle\langle\mathtt{a}\rangle\langle\mathtt{b}\rangle\langle\mathtt{b}\rangle\varphi) \vee \cdots$. Thus, the formula means that there is a transition sequence of the form $\mathtt{a}^n\mathtt{b}^n$ for some $n \geq 0$ that leads to a state satisfying φ.

Following [19], we exclude out unmeaningful formulas such as $(\langle a\rangle\mathbf{true})+1$ by using a simple type system. The types $\bullet$, $\mathtt{int}$, and $\sigma \to \tau$ describe propositions, integers, and (monotonic) functions from σ to τ, respectively. Note that the integer type $\mathtt{int}$ may occur only in an argument position; this restriction is required to ensure that least and greatest fixpoints are well-defined. The typing rules for formulas are given in Fig. 2. In the figure, Δ denotes a type environment, which is a finite map from variables to (extended) types. Below we consider only well-typed formulas.

2.2 Semantics and HFL$_\mathbf{Z}$ Model Checking

We now define the formal semantics of HFL$_\mathbf{Z}$ formulas. A *labeled transition system* (LTS) is a quadruple $\mathtt{L} = (U, A, \longrightarrow, \mathtt{s}_{\mathtt{init}})$, where U is a finite set of states, A is a finite set of actions, $\longrightarrow\ \subseteq U \times A \times U$ is a labeled transition relation, and $\mathtt{s}_{\mathtt{init}} \in U$ is the initial state. We write $\mathtt{s}_1 \xrightarrow{a} \mathtt{s}_2$ when $(\mathtt{s}_1, a, \mathtt{s}_2) \in\ \longrightarrow$.

For an LTS $\mathtt{L} = (U, A, \longrightarrow, \mathtt{s}_{\mathtt{init}})$ and an extended type σ, we define the partially ordered set $(\mathcal{D}_{\mathtt{L},\sigma}, \sqsubseteq_{\mathtt{L},\sigma})$ inductively by:

$$\begin{aligned}&\mathcal{D}_{\mathtt{L},\bullet} = 2^U \qquad \sqsubseteq_{\mathtt{L},\bullet} = \subseteq \qquad \mathcal{D}_{\mathtt{L},\mathtt{int}} = \mathbf{Z} \qquad \sqsubseteq_{\mathtt{L},\mathtt{int}} = \{(n,n) \mid n \in \mathbf{Z}\}\\ &\mathcal{D}_{\mathtt{L},\sigma\to\tau} = \{f \in \mathcal{D}_{\mathtt{L},\sigma} \to \mathcal{D}_{\mathtt{L},\tau} \mid \forall x, y.(x \sqsubseteq_{\mathtt{L},\sigma} y \Rightarrow f\,x \sqsubseteq_{\mathtt{L},\tau} f\,y)\}\\ &\sqsubseteq_{\mathtt{L},\sigma\to\tau} = \{(f,g) \mid \forall x \in \mathcal{D}_{\mathtt{L},\sigma}.f(x) \sqsubseteq_{\mathtt{L},\tau} g(x)\}\end{aligned}$$

$$\frac{}{\Delta \vdash_{\mathtt{H}} n : \mathtt{int}} \quad \text{(HT-Int)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \mathtt{int} \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1 \mathbin{\mathtt{op}} \varphi_2 : \mathtt{int}} \quad \text{(HT-Op)}$$

$$\frac{}{\Delta \vdash_{\mathtt{H}} \mathbf{true} : \bullet} \quad \text{(HT-True)}$$

$$\frac{}{\Delta \vdash_{\mathtt{H}} \mathbf{false} : \bullet} \quad \text{(HT-False)}$$

$$\frac{\mathtt{arity}(p) = k \qquad \Delta \vdash_{\mathtt{H}} \varphi_i : \mathtt{int} \text{ for each } i \in \{1,\dots,k\}}{\Delta \vdash_{\mathtt{H}} p(\varphi_1,\dots,\varphi_k) : \bullet} \quad \text{(HT-Pred)}$$

$$\frac{}{\Delta, X : \sigma \vdash_{\mathtt{H}} X : \sigma} \quad \text{(HT-Var)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \bullet \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1 \vee \varphi_2 : \bullet} \quad \text{(HT-Or)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \bullet \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1 \wedge \varphi_2 : \bullet} \quad \text{(HT-And)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi : \bullet}{\Delta \vdash_{\mathtt{H}} \langle a \rangle \varphi : \bullet} \quad \text{(HT-Some)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi : \bullet}{\Delta \vdash_{\mathtt{H}} [a] \varphi : \bullet} \quad \text{(HT-All)}$$

$$\frac{\Delta, X : \tau \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \mu X^{\tau}.\, \varphi : \tau} \quad \text{(HT-Mu)}$$

$$\frac{\Delta, X : \tau \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \nu X^{\tau}.\, \varphi : \tau} \quad \text{(HT-Nu)}$$

$$\frac{\Delta, X : \sigma \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \lambda X : \sigma.\, \varphi : \sigma \to \tau} \quad \text{(HT-Abs)}$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_1 : \sigma \to \tau \quad \Delta \vdash_{\mathtt{H}} \varphi_2 : \sigma}{\Delta \vdash_{\mathtt{H}} \varphi_1\ \varphi_2 : \tau} \quad \text{(HT-App)}$$

Fig. 2. Typing rules for $\mathrm{HFL}_{\mathbf{Z}}$ formulas

Note that $(\mathcal{D}_{\mathtt{L},\tau}, \sqsubseteq_{\mathtt{L},\tau})$ forms a complete lattice (but $(\mathcal{D}_{\mathtt{L},\mathtt{int}}, \sqsubseteq_{\mathtt{L},\mathtt{int}})$ does not). We write $\bot_{\mathtt{L},\tau}$ and $\top_{\mathtt{L},\tau}$ for the least and greatest elements of $\mathcal{D}_{\mathtt{L},\tau}$ (which are $\lambda\widetilde{x}.\emptyset$ and $\lambda\widetilde{x}.U$) respectively. We sometimes omit the subscript $\mathtt{L}$ below. Let $[\![\Delta]\!]_{\mathtt{L}}$ be the set of functions (called *valuations*) that maps X to an element of $\mathcal{D}_{\mathtt{L},\sigma}$ for each $X : \sigma \in \Delta$. For an HFL formula φ such that $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$, we define $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}$ as a map from $[\![\Delta]\!]_{\mathtt{L}}$ to $\mathcal{D}_\sigma$, by induction on the derivation[4] of $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$, as follows.

$$[\![\Delta \vdash_{\mathtt{H}} n : \mathtt{int}]\!]_{\mathtt{L}}(\rho) = n \qquad [\![\Delta \vdash_{\mathtt{H}} \mathbf{true} : \bullet]\!]_{\mathtt{L}}(\rho) = U \qquad [\![\Delta \vdash_{\mathtt{H}} \mathbf{false} : \bullet]\!]_{\mathtt{L}}(\rho) = \emptyset$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1 \mathbin{\mathtt{op}} \varphi_2 : \mathtt{int}]\!]_{\mathtt{L}}(\rho) = ([\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \mathtt{int}]\!]_{\mathtt{L}}(\rho))[\![\mathtt{op}]\!]([\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \mathtt{int}]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} p(\varphi_1,\dots,\varphi_k) : \bullet]\!]_{\mathtt{L}}(\rho) =$$
$$\begin{cases} U \text{ if } ([\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \mathtt{int}]\!]_{\mathtt{L}}(\rho),\dots,[\![\Delta \vdash_{\mathtt{H}} \varphi_k : \mathtt{int}]\!]_{\mathtt{L}}(\rho)) \in [\![p]\!] \\ \emptyset \ \text{ otherwise} \end{cases}$$

$$[\![\Delta, X : \sigma \vdash_{\mathtt{H}} X : \sigma]\!]_{\mathtt{L}}(\rho) = \rho(X)$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1 \vee \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \bullet]\!]_{\mathtt{L}}(\rho) \cup [\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho)$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1 \wedge \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \bullet]\!]_{\mathtt{L}}(\rho) \cap [\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho)$$

$$[\![\Delta \vdash_{\mathtt{H}} \langle a \rangle \varphi : \bullet]\!]_{\mathtt{L}}(\rho) = \{\mathtt{s} \mid \exists \mathtt{s}' \in [\![\Delta \vdash_{\mathtt{H}} \varphi : \bullet]\!]_{\mathtt{L}}(\rho).\ \mathtt{s} \xrightarrow{a} \mathtt{s}'\}$$

$$[\![\Delta \vdash_{\mathtt{H}} [a] \varphi : \bullet]\!]_{\mathtt{L}}(\rho) = \{\mathtt{s} \mid \forall \mathtt{s}' \in U.\ (\mathtt{s} \xrightarrow{a} \mathtt{s}' \text{ implies } \mathtt{s}' \in [\![\Delta \vdash_{\mathtt{H}} \varphi : \bullet]\!]_{\mathtt{L}}(\rho))\}$$

$$[\![\Delta \vdash_{\mathtt{H}} \mu X^{\tau}.\varphi : \tau]\!]_{\mathtt{L}}(\rho) = \mathbf{lfp}_{\mathtt{L},\tau}([\![\Delta \vdash_{\mathtt{H}} \lambda X : \tau.\ \varphi : \tau \to \tau]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} \nu X^{\tau}.\varphi : \tau]\!]_{\mathtt{L}}(\rho) = \mathbf{gfp}_{\mathtt{L},\tau}([\![\Delta \vdash_{\mathtt{H}} \lambda X : \tau.\ \varphi : \tau \to \tau]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} \lambda X : \sigma.\ \varphi : \sigma \to \tau]\!]_{\mathtt{L}}(\rho) = \{(v, [\![\Delta, X : \sigma \vdash_{\mathtt{H}} \varphi : \tau]\!]_{\mathtt{L}}(\rho[X \mapsto v])) \mid v \in \mathcal{D}_{\mathtt{L},\sigma}\}$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1\ \varphi_2 : \tau]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \sigma \to \tau]\!]_{\mathtt{L}}(\rho)([\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \sigma]\!]_{\mathtt{L}}(\rho))$$

[4] Note that the derivation of each judgment $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$ is unique if there is any.

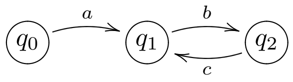
Here, $[\![\mathtt{op}]\!]$ denotes the binary function on integers represented by $\mathtt{op}$ and $[\![p]\!]$ denotes the k-ary relation on integers represented by p. The least/greatest fixpoint operators $\mathbf{lfp}_{\mathtt{L},\tau}$ and $\mathbf{gfp}_{\mathtt{L},\tau}$ are defined by $\mathbf{lfp}_{\mathtt{L},\tau}(f) = \bigsqcap_{\mathtt{L},\tau}\{x \in \mathcal{D}_{\mathtt{L},\tau} \mid f(x) \sqsubseteq_{\mathtt{L},\tau} x\}$ and $\mathbf{gfp}_{\mathtt{L},\tau}(f) = \bigsqcup_{\mathtt{L},\tau}\{x \in \mathcal{D}_{\mathtt{L},\tau} \mid x \sqsubseteq_{\mathtt{L},\tau} f(x)\}$. Here, $\bigsqcup_{\mathtt{L},\tau}$ and $\bigsqcap_{\mathtt{L},\tau}$ respectively denote the least upper bound and the greatest lower bound with respect to $\sqsubseteq_{\mathtt{L},\tau}$. We often omit the subscript $\mathtt{L}$ and write $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]$ for $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}$. For a closed formula, i.e., a formula well-typed under the empty type environment $\emptyset$, we often write $[\![\varphi]\!]_{\mathtt{L}}$ or just $[\![\varphi]\!]$ for $[\![\emptyset \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}(\emptyset)$.

Example 2. For the LTS $\mathtt{L}_{\mathit{file}}$ in Fig. 1, we have:

$$\begin{aligned}&[\![\nu X^{\bullet}.(\langle\mathtt{close}\rangle\mathbf{true} \wedge \langle\mathtt{read}\rangle X)]\!] =\\ &\quad \mathbf{gfp}_{\mathtt{L},\bullet}(\lambda x \in \mathcal{D}_{\mathtt{L},\bullet}.[\![X : \bullet \vdash \langle\mathtt{close}\rangle\mathbf{true} \wedge \langle\mathtt{read}\rangle X : \bullet]\!](\{X \mapsto x\})) = \{q_0\}.\end{aligned}$$

In fact, $x = \{q_0\} \in \mathcal{D}_{\mathtt{L},\bullet}$ satisfies the equation: $[\![X : \bullet \vdash \langle\mathtt{close}\rangle\mathbf{true} \wedge \langle\mathtt{read}\rangle X : \bullet]\!]_{\mathtt{L}}(\{X \mapsto x\}) = x$, and $x = \{q_0\} \in \mathcal{D}_{\mathtt{L},\bullet}$ is the greatest such element.

Consider the following LTS $\mathtt{L}_1$:


$$\begin{tikzcd} q_0 \arrow[r, "a"] & q_1 \arrow[r, bend left, "b"] & q_2 \arrow[l, bend left, "c"] \end{tikzcd}$$

and $\varphi_{\mathtt{ab}}\,(\langle c\rangle\mathbf{true})$ where $\varphi_{\mathtt{ab}}$ is the one introduced in Example 1. Then, $[\![\varphi_{\mathtt{ab}}\,(\langle c\rangle\mathbf{true})]\!]_{\mathtt{L}_1} = \{q_0, q_2\}$.

Definition 1 (HFL$_{\mathbf{Z}}$ model checking). *For a closed formula φ of type $\bullet$, we write* $\mathtt{L}, \mathtt{s} \models \varphi$ *if* $\mathtt{s} \in [\![\varphi]\!]_{\mathtt{L}}$*, and write* $\mathtt{L} \models \varphi$ *if* $\mathtt{s}_{\mathtt{init}} \in [\![\varphi]\!]_{\mathtt{L}}$. HFL$_{\mathbf{Z}}$ model checking *is the problem of, given* $\mathtt{L}$ *and* φ*, deciding whether* $\mathtt{L} \models \varphi$ *holds.*

The HFL$_{\mathbf{Z}}$ model checking problem is *un*decidable, due to the presence of integers; in fact, the semantic domain $\mathcal{D}_{\mathtt{L},\sigma}$ is not finite for σ that contains $\mathtt{int}$. The undecidability is obtained as a corollary of the soundness and completeness of the reduction from the may-reachability problem to HFL model checking discussed in Sect. 5. For the fragment of pure HFL (i.e., HFL$_{\mathbf{Z}}$ without integers, which we write HFL$_{\emptyset}$ below), the model checking problem is decidable [42].

The *order* of an HFL$_{\mathbf{Z}}$ model checking problem $\mathtt{L} \overset{?}{\models} \varphi$ is the highest order of types of subformulas of φ, where the order of a type is defined by: $\mathtt{order}(\bullet) = \mathtt{order}(\mathtt{int}) = 0$ and $\mathtt{order}(\sigma \to \tau) = \max(\mathtt{order}(\sigma) + 1, \mathtt{order}(\tau))$. The complexity of order-k HFL$_{\emptyset}$ model checking is k-EXPTIME complete [1], but polynomial time in the size of HFL formulas under the assumption that the other parameters (the size of LTS and the largest size of types used in formulas) are fixed [19].

Remark 1. Though we do not have quantifiers on integers as primitives, we can encode them using fixpoint operators. Given a formula $\varphi : \mathtt{int} \to \bullet$, we can express $\exists x : \mathtt{int}.\varphi(x)$ and $\forall x : \mathtt{int}.\varphi(x)$ by $(\mu X^{\mathtt{int}\to\bullet}.\lambda x : \mathtt{int}.\varphi(x) \vee X(x-1) \vee X(x+1))0$ and $(\nu X^{\mathtt{int}\to\bullet}.\lambda x : \mathtt{int}.\varphi(x) \wedge X(x-1) \wedge X(x+1))0$ respectively.

2.3 HES

As in [19], we often write an HFL$_\mathbf{Z}$ formula as a sequence of fixpoint equations, called a *hierarchical equation system* (HES).

Definition 2. *An (extended)* hierarchical equation system *(HES) is a pair* $(\mathcal{E}, \varphi)$ *where* $\mathcal{E}$ *is a sequence of fixpoint equations, of the form:* $X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n$*, where* $\alpha_i \in \{\mu, \nu\}$*. We assume that* $X_1 : \tau_1, \ldots, X_n : \tau_n \vdash_{\mathtt{H}} \varphi_i : \tau_i$ *holds for each* $i \in \{1, \ldots, n\}$*, and that* $\varphi_1, \ldots, \varphi_n, \varphi$ *do not contain any fixpoint operators.*

The HES $\Phi = (\mathcal{E}, \varphi)$ represents the HFL$_\mathbf{Z}$ formula $\mathit{toHFL}(\mathcal{E}, \varphi)$ defined inductively by: $\mathit{toHFL}(\epsilon, \varphi) = \varphi$ and $\mathit{toHFL}(\mathcal{E}; X^\tau =_\alpha \varphi', \varphi) = \mathit{toHFL}([\alpha X^\tau.\varphi'/X]\mathcal{E}, [\alpha X^\tau.\varphi'/X]\varphi)$. Conversely, every HFL$_\mathbf{Z}$ formula can be easily converted to an equivalent HES. In the rest of the paper, we often represent an HFL$_\mathbf{Z}$ formula in the form of HES, and just call it an HFL$_\mathbf{Z}$ formula. We write $[\![\Phi]\!]$ for $[\![\mathit{toHFL}(\Phi)]\!]$. An HES $(X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n, \varphi)$ can be normalized to $(X_0^{\tau_0} =_\nu \varphi; X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n, X_0)$ where τ_0 is the type of φ. Thus, we sometimes call just a sequence of equations $X_0^{\tau_0} =_\nu \varphi; X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n$ an HES, with the understanding that "the main formula" is the first variable X_0. Also, we often write $X^\tau\ x_1 \cdots x_k =_\alpha \varphi$ for the equation $X^\tau =_\alpha \lambda x_1. \cdots \lambda x_k.\varphi$. We often omit type annotations and just write $X =_\alpha \varphi$ for $X^\tau =_\alpha \varphi$.

Example 3. The formula $\nu X.\mu Y.\langle\mathtt{b}\rangle X \vee \langle\mathtt{a}\rangle Y$ (which means that the current state has a transition sequence of the form $(\mathtt{a}^*\mathtt{b})^\omega$) is expressed as the following HES:

$$((X =_\nu Y; Y =_\mu \langle\mathtt{b}\rangle X \vee \langle\mathtt{a}\rangle Y), \quad X).$$

3 Warming Up

To help readers get more familiar with HFL$_\mathbf{Z}$ and the idea of reductions, we give here some variations of the examples of verification of file-accessing programs in Sect. 1, which are instances of the "resource usage verification problem" [15]. General reductions will be discussed in Sects. 5, 6 and 7, after the target language is set up in Sect. 4.

Consider the following OCaml-like program, which uses exceptions.

```
let readex x = read x; (if * then () else raise Eof) in
let rec f x = readex x; f x in
let d = open_in "foo" in try f d with Eof -> close d
```

Here, `*` represents a non-deterministic boolean value. The function `readex` reads the file pointer x, and then non-deterministically raises an end-of-file (`Eof`) exception. The main expression (on the third line) first opens file "foo", calls `f` to read the file repeatedly, and closes the file upon an end-of-file exception. Suppose, as in the example of Sect. 1, we wish to verify that the file "foo" is accessed following the protocol in Fig. 1.

First, we can remove exceptions by representing an exception handler as a special continuation [6]:

```
let readex x h k = read x; (if * then k() else h()) in
let rec f x h k = readex x h (fun _ -> f x h k) in
let d = open_in "foo" in f d (fun _ -> close d) (fun _ -> ())
```

Here, we have added to each function two parameters `h` and `k`, which represent an exception handler and a (normal) continuation respectively.

Let Φ be $(\mathcal{E}, F\ \mathbf{true}\ (\lambda r.\langle \mathtt{close}\rangle \mathbf{true})\ (\lambda r.\mathbf{true}))$ where $\mathcal{E}$ is:

$$\begin{aligned}&\mathit{Readex}\ x\ h\ k =_\nu \langle \mathtt{read}\rangle(k\,\mathbf{true} \land h\,\mathbf{true});\\ &F\ x\ h\ k =_\nu \mathit{Readex}\ x\ h\ (\lambda r.F\ x\ h\ k).\end{aligned}$$

Here, we have just replaced read/close operations with the modal operators $\langle \mathtt{read}\rangle$ and $\langle \mathtt{close}\rangle$, non-deterministic choice with a logical conjunction, and the unit value ( ) with **true**. Then, $\mathtt{L}_{\mathit{file}} \models \Phi$ if and only if the program performs only valid accesses to the file (e.g., it does not access the file after a close operation), where $\mathtt{L}_{\mathit{file}}$ is the LTS shown in Fig. 1. The correctness of the reduction can be informally understood by observing that there is a close correspondence between reductions of the program and those of the HFL formula above, and when the program reaches a read command $\mathtt{read}\ x$, the corresponding formula is of the form $\langle \mathtt{read}\rangle \cdots$, meaning that the read operation is valid in the current state; a similar condition holds also for close operations. We will present a general translation and prove its correctness in Sect. 6.

Let us consider another example, which uses integers:

```
let rec f y x k = if y=0 then (close x; k())
                  else (read x; f (y-1) x k) in
let d = open_in "foo" in f n d (fun _ -> ())
```

Here, `n` is an integer constant. The function `f` reads `x` `y` times, and then calls the continuation `k`. Let $\mathtt{L}'_{\mathit{file}}$ be the LTS obtained by adding to $\mathtt{L}_{\mathit{file}}$ a new state q_2 and the transition $q_1 \xrightarrow{\mathtt{end}} q_2$ (which intuitively means that a program is allowed to terminate in the state q_1), and let Φ' be $(\mathcal{E}', F\ n\ \mathbf{true}\ (\lambda r.\langle \mathtt{end}\rangle \mathbf{true}))$ where $\mathcal{E}'$ is:

$$F\ y\ x\ k =_\mu (y = 0 \Rightarrow \langle \mathtt{close}\rangle(k\,\mathbf{true})) \land (y \neq 0 \Rightarrow \langle \mathtt{read}\rangle(F\ (y-1)\ x\ k)).$$

Here, $p(\varphi_1, \dots, \varphi_k) \Rightarrow \varphi$ is an abbreviation of $\neg p(\varphi_1, \dots, \varphi_k) \lor \varphi$. Then, $\mathtt{L}'_{\mathit{file}} \models \Phi'$ if and only if (i) the program performs only valid accesses to the file, (ii) it eventually terminates, and (iii) the file is closed when the program terminates. Notice the use of μ instead of ν above; by using μ, we can express liveness properties. The property $\mathtt{L}'_{\mathit{file}} \models \Phi'$ indeed holds for $n \geq 0$, but not for $n < 0$. In fact, $F\ n\ x\ k$ is equivalent to **false** for $n < 0$, and $\langle \mathtt{read}\rangle^n \langle \mathtt{close}\rangle(k\ \mathbf{true})$ for $n \geq 0$.

4 Target Language

This section sets up, as the target of program verification, a call-by-name[5] higher-order functional language extended with events. The language is essentially the same as the one used by Watanabe et al. [43] for discussing fair non-termination.

4.1 Syntax and Typing

We assume a finite set **Ev** of names called *events*, ranged over by a, and a denumerable set of variables, ranged over by $x, y, \ldots$. Events are used to express temporal properties of programs. We write $\widetilde{x}$ ($\widetilde{t}$, resp.) for a sequence of variables (terms, resp.), and write $|\widetilde{x}|$ for the length of the sequence.

A *program* is a pair (D, t) consisting of a set D of function definitions $\{f_1\ \widetilde{x}_1 = t_1, \ldots, f_n\ \widetilde{x}_n = t_n\}$ and a term t. The set of *terms*, ranged over by t, is defined by:

$$t ::= (\,) \mid x \mid n \mid t_1\ \mathtt{op}\ t_2 \mid \mathbf{event}\ a; t \mid \mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\ \mid t_1 t_2 \mid t_1 \Box t_2.$$

Here, n and p range over the sets of integers and integer predicates as in HFL formulas. The expression **event** $a; t$ raises an event a, and then evaluates t. Events are used to encode program properties of interest. For example, an assertion $\mathbf{assert}(b)$ can be expressed as **if** b **then** $(\,)$ **else** $(\mathbf{event}\ \mathtt{fail}; \Omega)$, where `fail` is an event that expresses an assertion failure and Ω is a non-terminating term. If program termination is of interest, one can insert "**event** `end`" to every termination point and check whether an `end` event occurs. The expression $t_1 \Box t_2$ evaluates t_1 or t_2 in a non-deterministic manner; it can be used to model, e.g., unknown inputs from an environment. We use the meta-variable P for programs. When $P = (D, t)$ with $D = \{f_1\ \widetilde{x}_1 = t_1, \ldots, f_n\ \widetilde{x}_n = t_n\}$, we write $\mathbf{funs}(P)$ for $\{f_1, \ldots, f_n\}$ (i.e., the set of function names defined in P). Using λ-abstractions, we sometimes write $f = \lambda\widetilde{x}.t$ for the function definition $f\ \widetilde{x} = t$. We also regard D as a map from function names to terms, and write $dom(D)$ for $\{f_1, \ldots, f_n\}$ and $D(f_i)$ for $\lambda\widetilde{x}_i.t_i$.

Any program (D, t) can be normalized to $(D \cup \{\mathbf{main} = t\}, \mathbf{main})$ where **main** is a name for the "main" function. We sometimes write just D for a program $(D, \mathbf{main})$, with the understanding that D contains a definition of **main**.

We restrict the syntax of expressions using a type system. The set of *simple types*, ranged over by κ, is defined by:

$$\kappa ::= \star \mid \eta \to \kappa \qquad\qquad \eta ::= \kappa \mid \mathtt{int}.$$

The types $\star$, `int`, and $\eta \to \kappa$ describe the unit value, integers, and functions from η to κ respectively. Note that `int` is allowed to occur only in argument

[5] Call-by-value programs can be handled by applying the CPS transformation before applying the reductions to HFL model checking.

positions. We defer typing rules to [23], as they are standard, except that we require that the righthand side of each function definition must have type $\star$; this restriction, as well as the restriction that `int` occurs only in argument positions, does not lose generality, as those conditions can be ensured by applying CPS transformation. We consider below only well-typed programs.

4.2 Operational Semantics

We define the labeled transition relation $t \xrightarrow{\ell}_D t'$, where ℓ is either ϵ or an event name, as the least relation closed under the rules in Fig. 3. We implicitly assume that the program (D, t) is well-typed, and this assumption is maintained throughout reductions by the standard type preservation property. In the rules for if-expressions, $[\![t'_i]\!]$ represents the integer value denoted by t'_i; note that the well-typedness of (D, t) guarantees that t'_i must be arithmetic expressions consisting of integers and integer operations; thus, $[\![t'_i]\!]$ is well defined. We often omit the subscript D when it is clear from the context. We write $t \xrightarrow{\ell_1 \cdots \ell_k}{}^{*}_D t'$ if $t \xrightarrow{\ell_1}_D \cdots \xrightarrow{\ell_k}_D t'$. Here, ϵ is treated as an empty sequence; thus, for example, we write $t \xrightarrow{ab}{}^{*}_D t'$ if $t \xrightarrow{a}_D \xrightarrow{\epsilon}_D \xrightarrow{b}_D \xrightarrow{\epsilon}_D t'$.

$$\frac{}{\textbf{event } a; t \xrightarrow{a}_D t} \qquad \frac{f\,\widetilde{x} = u \in D \qquad |\widetilde{x}| = |\widetilde{t}|}{f\,\widetilde{t} \xrightarrow{\epsilon}_D [\widetilde{t}/\widetilde{x}]u} \qquad \frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \in [\![p]\!]}{\textbf{if } p(t'_1, \ldots, t'_k) \textbf{ then } t_1 \textbf{ else } t_2 \xrightarrow{\epsilon}_D t_1}$$

$$\frac{i \in \{1, 2\}}{t_1 \Box t_2 \xrightarrow{\epsilon}_D t_i} \qquad \frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \notin [\![p]\!]}{\textbf{if } p(t'_1, \ldots, t'_k) \textbf{ then } t_1 \textbf{ else } t_2 \xrightarrow{\epsilon}_D t_2}$$

Fig. 3. Labeled transition semantics

For a program $P = (D, t_0)$, we define the set $\textbf{Traces}(P) (\subseteq \textbf{Ev}^* \cup \textbf{Ev}^\omega)$ of *traces* by:

$$\begin{aligned}\textbf{Traces}(D, t_0) = \{&\ell_0 \cdots \ell_{n-1} \in (\{\epsilon\} \cup \textbf{Ev})^* \mid \forall i \in \{0, \ldots, n-1\}. t_i \xrightarrow{\ell_i}_D t_{i+1}\} \\ \cup\{&\ell_0 \ell_1 \cdots \in (\{\epsilon\} \cup \textbf{Ev})^\omega \mid \forall i \in \omega. t_i \xrightarrow{\ell_i}_D t_{i+1}\}.\end{aligned}$$

Note that since the label ϵ is regarded as an empty sequence, $\ell_0\ell_1\ell_2 = aa$ if $\ell_0 = \ell_2 = a$ and $\ell_1 = \epsilon$, and an element of $(\{\epsilon\} \cup \textbf{Ev})^\omega$ is regarded as that of $\textbf{Ev}^* \cup \textbf{Ev}^\omega$. We write $\textbf{FinTraces}(P)$ and $\textbf{InfTraces}(P)$ for $\textbf{Traces}(P) \cap \textbf{Ev}^*$ and $\textbf{Traces}(P) \cap \textbf{Ev}^\omega$ respectively. The set of *full traces* $\textbf{FullTraces}(D, t_0) (\subseteq \textbf{Ev}^* \cup \textbf{Ev}^\omega)$ is defined as:

$$\begin{aligned}\{&\ell_0 \cdots \ell_{n-1} \in (\{\epsilon\} \cup \textbf{Ev})^* \mid t_n = (\,) \wedge \forall i \in \{0, \ldots, n-1\}. t_i \xrightarrow{\ell_i}_D t_{i+1}\} \\ \cup\{&\ell_0 \ell_1 \cdots \in (\{\epsilon\} \cup \textbf{Ev})^\omega \mid \forall i \in \omega. t_i \xrightarrow{\ell_i}_D t_{i+1}\}.\end{aligned}$$

Example 4. The last example in Sect. 1 is modeled as $P_{\mathit{file}} = (D, f\,())$, where $D = \{f\,x = (\textbf{event}\ \texttt{close}; ())\Box(\textbf{event}\ \texttt{read}; \textbf{event}\ \texttt{read}; f\,x)\}$. We have:

$$\begin{array}{l}
\textbf{Traces}(P) = \{\texttt{read}^n \mid n \geq 0\} \cup \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\} \cup \{\texttt{read}^\omega\}\\
\textbf{FinTraces}(P) = \{\texttt{read}^n \mid n \geq 0\} \cup \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\}\\
\textbf{InfTraces}(P) = \{\texttt{read}^\omega\}\ \textbf{FullTraces}(P) = \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\} \cup \{\texttt{read}^\omega\}.
\end{array}$$

5 May/Must-Reachability Verification

Here we consider the following problems:

- May-reachability: "Given a program P and an event a, may P raise a?"
- Must-reachability: "Given a program P and an event a, must P raise a?"

Since we are interested in a particular event a, we restrict here the event set **Ev** to a singleton set of the form $\{a\}$. Then, the may-reachability is formalized as $a \overset{?}{\in} \textbf{Traces}(P)$, whereas the must-reachability is formalized as "does every trace in $\textbf{FullTraces}(P)$ contain a?" We encode both problems into the validity of $\text{HFL}_{\mathbf{Z}}$ formulas (without any modal operators $\langle a\rangle$ or $[a]$), or the $\text{HFL}_{\mathbf{Z}}$ model checking of those formulas against a trivial model (which consists of a single state without any transitions). Since our reductions are sound and complete, the characterizations of their negations –non-reachability and may-non-reachability– can also be obtained immediately. Although these are the simplest classes of properties among those discussed in Sects. 5, 6 and 7, they are already large enough to accommodate many program properties discussed in the literature, including lack of assertion failures/uncaught exceptions [22] (which can be characterized as non-reachability; recall the encoding of assertions in Sect. 4), termination [27,29] (characterized as must-reachability), and non-termination [26] (characterized as may-non-reachability).

5.1 May-Reachability

As in the examples in Sect. 3, we translate a program to a formula that says "the program may raise an event a" in a compositional manner. For example, **event** $a; t$ can be translated to **true** (since the event will surely be raised immediately), and $t_1 \Box t_2$ can be translated to $t_1^\dagger \vee t_2^\dagger$ where $t_i^\dagger$ is the result of the translation of t_i (since only one of t_1 and t_2 needs to raise an event).

Definition 3. *Let $P = (D, t)$ be a program. $\Phi_{P,\mathit{may}}$ is the HES $(D^{\dagger_{\mathit{may}}}, t^{\dagger_{\mathit{may}}})$, where $D^{\dagger_{\mathit{may}}}$ and $t^{\dagger_{\mathit{may}}}$ are defined by:*

$$\begin{array}{l}
\{f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{\mathit{may}}} = \left(f_1\,\widetilde{x}_1 =_\mu t_1{}^{\dagger_{\mathit{may}}}; \cdots; f_n\,\widetilde{x}_n =_\mu t_n{}^{\dagger_{\mathit{may}}}\right)\\
(\,)^{\dagger_{\mathit{may}}} = \textbf{false} \qquad x^{\dagger_{\mathit{may}}} = x \qquad n^{\dagger_{\mathit{may}}} = n \qquad (t_1\ \mathsf{op}\ t_2)^{\dagger_{\mathit{may}}} = t_1{}^{\dagger_{\mathit{may}}}\ \mathsf{op}\ t_2{}^{\dagger_{\mathit{may}}}\\
(\textbf{if}\ p(t'_1, \ldots, t'_k)\ \textbf{then}\ t_1\ \textbf{else}\ t_2)^{\dagger_{\mathit{may}}} =\\
\qquad\qquad (p(t'_1{}^{\dagger_{\mathit{may}}}, \ldots, t'_k{}^{\dagger_{\mathit{may}}}) \wedge t_1{}^{\dagger_{\mathit{may}}}) \vee (\neg p(t'_1{}^{\dagger_{\mathit{may}}}, \ldots, t'_k{}^{\dagger_{\mathit{may}}}) \wedge t_2{}^{\dagger_{\mathit{may}}})\\
(\textbf{event}\ a; t)^{\dagger_{\mathit{may}}} = \textbf{true} \quad (t_1 t_2)^{\dagger_{\mathit{may}}} = t_1{}^{\dagger_{\mathit{may}}} t_2{}^{\dagger_{\mathit{may}}} \quad (t_1 \Box t_2)^{\dagger_{\mathit{may}}} = t_1{}^{\dagger_{\mathit{may}}} \vee t_2{}^{\dagger_{\mathit{may}}}.
\end{array}$$

Note that, in the definition of $D^{\dagger_{may}}$, the order of function definitions in D does not matter (i.e., the resulting HES is unique up to the semantic equality), since all the fixpoint variables are bound by μ.

Example 5. Consider the program:

$$P_{loop} = (\{loop\ x = loop\ x\}, loop(\textbf{event}\ a; ())).$$

It is translated to the HES $\Phi_{loop} = (loop\ x =_\mu loop\ x, loop(\textbf{true}))$. Since $loop \equiv \mu loop.\lambda x.loop\ x$ is equivalent to $\lambda x.\textbf{false}$, Φ_{loop} is equivalent to $\textbf{false}$. In fact, P_{loop} never raises an event a (recall that our language is call-by-name).

Example 6. Consider the program $P_{sum} = (D_{sum}, \textbf{main})$ where D_{sum} is:

$$\begin{array}{l}\textbf{main} = sum\ n\ (\lambda r.\textbf{assert}(r \geq n))\\ sum\ x\ k = \textbf{if}\ x = 0\ \textbf{then}\ k\,0\ \textbf{else}\ sum\ (x-1)\ (\lambda r.k(x+r))\end{array}$$

Here, n is some integer constant, and $\textbf{assert}(b)$ is the macro introduced in Sect. 4. We have used λ-abstractions for the sake of readability. The function *sum* is a CPS version of a function that computes the summation of integers from 1 to x. The main function computes the sum $r = 1 + \cdots + n$, and asserts $r \geq n$. It is translated to the HES $\Phi_{P_2,may} = (\mathcal{E}_{sum}, \textbf{main})$ where $\mathcal{E}_{sum}$ is:

$$\begin{array}{l}\textbf{main} =_\mu sum\ n\ (\lambda r.(r \geq n \wedge \textbf{false}) \vee (r < n \wedge \textbf{true}));\\ sum\ x\ k =_\mu (x = 0 \wedge k\,0) \vee (x \neq 0 \wedge sum\ (x-1)\ (\lambda r.k(x+r))).\end{array}$$

Here, n is treated as a constant. Since the shape of the formula does not depend on the value of n, the property "an assertion failure may occur for some n" can be expressed by $\exists n.\Phi_{P_2,may}$. □

The following theorem states that $\Phi_{P,may}$ is a complete characterization of the may-reachability of P.

Theorem 1. *Let P be a program. Then, $a \in \textbf{Traces}(P)$ if and only if $\mathtt{L}_0 \models \Phi_{P,may}$ for $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$.*

A proof of the theorem above is found in [23]. We only provide an outline. We first show the theorem for recursion-free programs and then lift it to arbitrary programs by using the continuity of functions represented in the fixpoint-free fragment of HFL$_{\mathbf{Z}}$ formulas. To show the theorem for recursion-free programs, we define the reduction relation $t \longrightarrow_D t'$ by:

$$\frac{f\widetilde{x} = u \in D \qquad |\widetilde{x}| = |\widetilde{t}|}{E[f\ \widetilde{t}] \longrightarrow_D E[[\widetilde{t}/\widetilde{x}]u]} \qquad \frac{(\llbracket t_1' \rrbracket, \ldots, \llbracket t_k' \rrbracket) \in \llbracket p \rrbracket}{E[\textbf{if}\ p(t_1', \ldots, t_k')\ \textbf{then}\ t_1\ \textbf{else}\ t_2] \longrightarrow_D E[t_1]}$$

$$\frac{(\llbracket t_1' \rrbracket, \ldots, \llbracket t_k' \rrbracket) \notin \llbracket p \rrbracket}{E[\textbf{if}\ p(t_1', \ldots, t_k')\ \textbf{then}\ t_1\ \textbf{else}\ t_2] \longrightarrow_D E[t_2]}$$

Here, E ranges over the set of evaluation contexts given by $E ::= [\,] \mid E \Box t \mid t \Box E \mid \textbf{event}\ a; E$. The reduction relation differs from the labeled transition relation given in Sect. 4, in that $\Box$ and **event** $a; \cdots$ are not eliminated. By the definition of the translation, the theorem holds for programs in normal form (with respect to the reduction relation), and the semantics of translated HFL formulas is preserved by the reduction relation; thus the theorem holds for recursion-free programs, as they are strongly normalizing.

5.2 Must-Reachability

The characterization of must-reachability can be obtained by an easy modification of the characterization of may-reachability: we just need to replace branches with logical conjunction.

Definition 4. *Let $P = (D, t)$ be a program. $\Phi_{P,must}$ is the HES $(D^{\dagger_{must}}, t^{\dagger_{must}})$, where $D^{\dagger_{must}}$ and $t^{\dagger_{must}}$ are defined by:*

$$\{f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{must}} = \left(f_1\;\widetilde{x}_1 =_\mu t_1{}^{\dagger_{must}}; \cdots; f_n\;\widetilde{x}_n =_\mu t_n{}^{\dagger_{must}}\right)$$
$$(\,)^{\dagger_{must}} = \mathbf{false} \qquad x^{\dagger_{must}} = x \qquad n^{\dagger_{must}} = n \qquad (t_1\;\mathtt{op}\;t_2)^{\dagger_{must}} = t_1{}^{\dagger_{must}}\;\mathtt{op}\;t_2{}^{\dagger_{must}}$$
$$(\mathbf{if}\ p(t_1', \ldots, t_k')\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\dagger_{must}} =$$
$$(p(t_1'{}^{\dagger_{must}}, \ldots, t_k'{}^{\dagger_{must}}) \Rightarrow t_1{}^{\dagger_{must}}) \wedge (\neg p(t_1'{}^{\dagger_{must}}, \ldots, t_k'{}^{\dagger_{must}}) \Rightarrow t_2{}^{\dagger_{must}})$$
$$(\mathbf{event}\ a; t)^{\dagger_{must}} = \mathbf{true} \quad (t_1 t_2)^{\dagger_{must}} = t_1{}^{\dagger_{must}} t_2{}^{\dagger_{must}} \;\; (t_1 \Box t_2)^{\dagger_{must}} = t_1{}^{\dagger_{must}} \wedge t_2{}^{\dagger_{must}}.$$

Here, $p(\varphi_1, \ldots, \varphi_k) \Rightarrow \varphi$ is a shorthand for $\neg p(\varphi_1, \ldots, \varphi_k) \vee \varphi$.

Example 7. Consider $P_{\mathtt{loop}} = (D, \mathtt{loop}\,m\,n)$ where D is:

$$\begin{aligned}\mathtt{loop}\;x\;y = \mathbf{if}\ x \le 0 \vee y \le 0\ \mathbf{then}\ (\mathbf{event}\ \mathtt{end}; (\,))\\ \mathbf{else}\ (\mathtt{loop}\;(x-1)\;(y*y))\Box(\mathtt{loop}\;x\;(y-1))\end{aligned}$$

Here, the event `end` is used to signal the termination of the program. The function `loop` non-deterministically updates the values of x and y until either x or y becomes non-positive. The must-termination of the program is characterized by $\Phi_{P_{\mathtt{loop}},must} = (\mathcal{E}, \mathtt{loop}\,m\,n)$ where $\mathcal{E}$ is:

$$\begin{aligned}\mathtt{loop}\;x\;y =_\mu\; & (x \le 0 \vee y \le 0 \Rightarrow \mathbf{true})\\ & \wedge(\neg(x \le 0 \vee y \le 0) \Rightarrow (\mathtt{loop}\;(x-1)\;(y*y)) \wedge (\mathtt{loop}\;x\;(y-1))).\end{aligned}$$

We write $\mathbf{Must}_a(P)$ if every $\pi \in \mathbf{FullTraces}(P)$ contains a. The following theorem, which can be proved in a manner similar to Theorem 1, guarantees that $\Phi_{P,must}$ is indeed a sound and complete characterization of the must-reachability.

Theorem 2. *Let P be a program. Then, $\boldsymbol{Must}_a(P)$ if and only if $\mathtt{L}_0 \models \Phi_{P,must}$ for $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$.*

6 Trace Properties

Here we consider the verification problem: "Given a (non-ω) regular language L and a program P, does *every* finite event sequence of P belong to L? (i.e. $\mathbf{FinTraces}(P) \overset{?}{\subseteq} L$)" and reduce it to an $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem. The verification of file-accessing programs considered in Sect. 3 may be considered an instance of the problem.

Here we assume that the language L is closed under the prefix operation; this does not lose generality because $\mathbf{FinTraces}(P)$ is also closed under the prefix operation. We write $A_L = (Q, \Sigma, \delta, q_0, F)$ for the minimal, deterministic automaton with no dead states (hence the transition function δ may be partial). Since L is prefix-closed and the automaton is minimal, $w \in L$ if and only if $\hat{\delta}(q_0, w)$ is defined (where $\hat{\delta}$ is defined by: $\hat{\delta}(q, \epsilon) = q$ and $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$). We use the corresponding LTS $\mathtt{L}_L = (Q, \Sigma, \{(q, a, q') \mid \delta(q, a) = q'\}, q_0)$ as the model of the reduced $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem.

Given the LTS $\mathtt{L}_L$ above, whether an event sequence $a_1 \cdots a_k$ belongs to L can be expressed as $\mathtt{L}_L \overset{?}{\models} \langle a_1 \rangle \cdots \langle a_k \rangle \mathbf{true}$. Whether all the event sequences in $\{a_{j,1} \cdots a_{j,k_j} \mid j \in \{1, \ldots, n\}\}$ belong to L can be expressed as $\mathtt{L}_L \overset{?}{\models} \bigwedge_{j \in \{1,\ldots,n\}} \langle a_{j,1} \rangle \cdots \langle a_{j,k_j} \rangle \mathbf{true}$. We can lift these translations for event sequences to the translation from a program (which can be considered a description of a set of event sequences) to an $\mathrm{HFL}_{\mathbf{Z}}$ formula, as follows.

Definition 5. *Let $P = (D, t)$ be a program. $\Phi_{P,path}$ is the HES $(D^{\dagger_{path}}, t^{\dagger_{path}})$, where $D^{\dagger_{path}}$ and $t^{\dagger_{path}}$ are defined by:*

$$\{f_1\, \widetilde{x}_1 = t_1, \ldots, f_n\, \widetilde{x}_n = t_n\}^{\dagger_{path}} = \left(f_1\, \widetilde{x}_1 =_\nu {t_1}^{\dagger_{path}}; \cdots ; f_n\, \widetilde{x}_n =_\nu {t_n}^{\dagger_{path}}\right)$$
$$(\,)^{\dagger_{path}} = \mathbf{true} \qquad x^{\dagger_{path}} = x \qquad n^{\dagger_{path}} = n \qquad (t_1\ \mathtt{op}\ t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}}\ \mathtt{op}\ {t_2}^{\dagger_{path}}$$
$$(\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\dagger_{path}} =$$
$$(p({t'_1}^{\dagger_{path}}, \ldots, {t'_k}^{\dagger_{path}}) \Rightarrow {t_1}^{\dagger_{path}}) \wedge (\neg p({t'_1}^{\dagger_{path}}, \ldots, {t'_k}^{\dagger_{path}}) \Rightarrow {t_2}^{\dagger_{path}})$$
$$(\mathbf{event}\ a; t)^{\dagger_{path}} = \langle a \rangle t^{\dagger_{path}} \quad (t_1 t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}} {t_2}^{\dagger_{path}} \ (t_1 \Box t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}} \wedge {t_2}^{\dagger_{path}}.$$

Example 8. The last program discussed in Sect. 3 is modeled as $P_2 = (D_2, f\ m\ g)$, where m is an integer constant and D_2 consists of:

$$f\ y\ k = \mathbf{if}\ y = 0\ \mathbf{then}\ (\mathbf{event}\ \mathtt{close}; k\,(\,))\ \mathbf{else}\ (\mathbf{event}\ \mathtt{read}; f\ (y-1)\ k)$$
$$g\ r = \mathbf{event}\ \mathtt{end}; (\,)$$

Here, we have modeled accesses to the file, and termination as events. Then, $\Phi_{P_2,path} = (\mathcal{E}_{P_2,path}, f\ m\ g)$ where $\mathcal{E}_{P_2,path}$ is:[6]

$$f\ n\ k =_\nu (n = 0 \Rightarrow \langle \mathtt{close} \rangle (k\, \mathbf{true})) \wedge (n \neq 0 \Rightarrow \langle \mathtt{read} \rangle (f\ (n-1)\ k))$$
$$g\ r =_\nu \langle \mathtt{end} \rangle \mathbf{true}.$$

Let L be the prefix-closure of $\mathtt{read}^* \cdot \mathtt{close} \cdot \mathtt{end}$. Then $\mathtt{L}_L$ is $\mathtt{L}'_{file}$ in Sect. 3, and $\mathbf{FinTraces}(P_2) \subseteq L$ can be verified by checking $\mathtt{L}_L \models \Phi_{P_2,path}$. □

[6] Unlike in Sect. 3, the variables are bound by ν since we are not concerned with the termination property here.

Theorem 3. *Let P be a program and L be a regular, prefix-closed language. Then,* $\mathbf{FinTraces}(P) \subseteq L$ *if and only if* $\mathtt{L}_L \models \Phi_{P,path}$.

As in Sect. 5, we first prove the theorem for programs in normal form, and then lift it to recursion-free programs by using the preservation of the semantics of HFL$_{\mathbf{Z}}$ formulas by reductions, and further to arbitrary programs by using the (co-)continuity of the functions represented by fixpoint-free HFL$_{\mathbf{Z}}$ formulas. See [23] for a concrete proof.

7 Linear-Time Temporal Properties

This section considers the following problem: "Given a program P and an ω-regular word language L, does $\mathbf{InfTraces}(P) \cap L = \emptyset$ hold?". From the viewpoint of program verification, L represents the set of "bad" behaviors. This can be considered an extension of the problems considered in the previous sections.

The reduction to HFL model checking is more involved than those in the previous sections. To see the difficulty, consider the program P_0:

$$(\{f = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f)\}, \quad f),$$

where c is some boolean expression. Let L be the complement of $(\mathtt{a}^*\mathtt{b})^\omega$, i.e., the set of infinite sequences that contain only finitely many $\mathtt{b}$'s. Following Sect. 6 (and noting that $\mathbf{InfTraces}(P) \cap L = \emptyset$ is equivalent to $\mathbf{InfTraces}(P) \subseteq (\mathtt{a}^*\mathtt{b})^\omega$ in this case), one may be tempted to prepare an LTS like the one in Fig. 4 (which corresponds to the transition function of a (parity) word automaton accepting $(\mathtt{a}^*\mathtt{b})^\omega$), and translate the program to an HES Φ_{P_0} of the form:

$$(f =_\alpha (c \Rightarrow \langle \mathtt{a} \rangle f) \wedge (\neg c \Rightarrow \langle \mathtt{b} \rangle f), \quad f),$$

where α is μ or ν. However, such a translation would not work. If $c = \mathbf{true}$, then $\mathbf{InfTraces}(P_0) = \mathtt{a}^\omega$, hence $\mathbf{InfTraces}(P_0) \cap L \neq \emptyset$; thus, α should be μ for Φ_{P_0} to be unsatisfied. If $c = \mathbf{false}$, however, $\mathbf{InfTraces}(P_0) = \mathtt{b}^\omega$, hence $\mathbf{InfTraces}(P_0) \cap L = \emptyset$; thus, α must be ν for Φ_{P_0} to be satisfied.

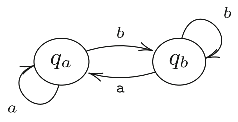


Fig. 4. LTS for $(a^*b)^\omega$

The example above suggests that we actually need to distinguish between the two occurrences of f in the body of f's definition. Note that in the then- and else-clauses respectively, f is called after different events $\mathtt{a}$ and $\mathtt{b}$. This difference is important, since we are interested in whether $\mathtt{b}$ occurs infinitely often. We thus duplicate f, and replace the program with the following program P_{dup}:

$$\begin{aligned}(\{f_b &= \textbf{if } c \textbf{ then } (\textbf{event } \mathtt{a}; f_a) \textbf{ else } (\textbf{event } \mathtt{b}; f_b),\\ f_a &= \textbf{if } c \textbf{ then } (\textbf{event } \mathtt{a}; f_a) \textbf{ else } (\textbf{event } \mathtt{b}; f_b)\}, f_b).\end{aligned}$$

For checking $\mathbf{InfTraces}(P_0) \cap L = \emptyset$, it is now sufficient to check that f_b is recursively called infinitely often. We can thus obtain the following HES:

$$((f_b =_\nu (c \Rightarrow \langle \mathtt{a}\rangle f_a) \wedge (\neg c \Rightarrow \langle \mathtt{b}\rangle f_b); \quad f_a =_\mu (c \Rightarrow \langle \mathtt{a}\rangle f_a) \wedge (\neg c \Rightarrow \langle \mathtt{b}\rangle f_b)), f_b).$$

Note that f_b and f_a are bound by ν and μ respectively, reflecting the fact that $\mathtt{b}$ should occur infinitely often, but $\mathtt{a}$ need not. If $c = \mathbf{true}$, the formula is equivalent to $\nu f_b.\langle \mathtt{a}\rangle \mu f_a.\langle \mathtt{a}\rangle f_a$, which is false. If $c = \mathbf{false}$, then the formula is equivalent to $\nu f_b.\langle \mathtt{b}\rangle f_b$, which is satisfied by by the LTS in Fig. 4.

The general translation is more involved due to the presence of higher-order functions, but, as in the example above, the overall translation consists of two steps. We first replicate functions according to what events may occur between two recursive calls, and reduce the problem $\mathbf{InfTraces}(P) \cap L \overset{?}{=} \emptyset$ to a problem of analyzing which functions are recursively called infinitely often, which we call a *call-sequence analysis*. We can then reduce the call-sequence analysis to HFL model checking in a rather straightforward manner (though the proof of the correctness is non-trivial). The resulting HFL formula actually does not contain modal operators.[7] So, as in Sect. 5, the resulting problem is the validity checking of HFL formulas without modal operators.

In the rest of this section, we first introduce the call-sequence analysis problem and its reduction to HFL model checking in Sect. 7.1. We then show how to reduce the temporal verification problem $\mathbf{InfTraces}(P) \cap L \overset{?}{=} \emptyset$ to an instance of the call-sequence analysis problem in Sect. 7.2.

7.1 Call-Sequence Analysis

We define the call-sequence analysis and reduce it to an HFL model-checking problem. As mentioned above, in the call-sequence analysis, we are interested in analyzing which functions are *recursively called* infinitely often. Here, we say that g is *recursively called from* f, if $f\,\widetilde{s} \xrightarrow{\epsilon}_D [\widetilde{s}/\widetilde{x}]t_f \xrightarrow{\widetilde{\ell}}{}^{*}_{D} g\,\widetilde{t}$, where $f\,\widetilde{x} = t_f \in D$ and g "originates from" t_f (a more formal definition will be given in Definition 6 below). For example, consider the following program P_{app}, which is a twisted version of P_{dup} above.

$$\begin{aligned}&(\{\mathtt{app}\, h\, x = h\, x,\\ &\quad f_b\, x = \textbf{if } x > 0 \textbf{ then } (\textbf{event } \mathtt{a}; \mathtt{app}\, f_a\, (x-1)) \textbf{ else } (\textbf{event } \mathtt{b}; \mathtt{app}\, f_b\, 5),\\ &\quad f_a\, x = \textbf{if } x > 0 \textbf{ then } (\textbf{event } \mathtt{a}; \mathtt{app}\, f_a\, (x-1)) \textbf{ else } (\textbf{event } \mathtt{b}; \mathtt{app}\, f_b\, 5)\}, f_b\, 5).\end{aligned}$$

[7] In the example above, we can actually remove $\langle \mathtt{a}\rangle$ and $\langle \mathtt{b}\rangle$, as information about events has been taken into account when f was duplicated.

Then f_a is "recursively called" from f_b in $f_b\,5 \xrightarrow{\mathsf{a}}_D^* \mathsf{app}\, f_a\, 4 \xrightarrow{\epsilon}_D^* f_a\, 4$ (and so is app). We are interested in infinite chains of recursive calls $f_0 f_1 f_2 \cdots$, and which functions may occur infinitely often in each chain. For instance, the program above has the unique infinite chain $(f_b f_a^5)^\omega$, in which both f_a and f_b occur infinitely often. (Besides the infinite chain, the program has finite chains like $f_b\,\mathsf{app}$; note that the chain cannot be extended further, as the body of app does not have any occurrence of recursive functions: app, f_a and f_b.)

We define the notion of "recursive calls" and call-sequences formally below.

Definition 6 (Recursive call relation, call sequences). *Let $P = (D, f_1\,\widetilde{s})$ be a program, with $D = \{f_i\,\widetilde{x}_i = u_i\}_{1\le i\le n}$. We define $D^\sharp := D \cup \{f_i^\sharp\,\widetilde{x} = u_i\}_{1\le i\le n}$ where $f_1^\sharp, \dots, f_n^\sharp$ are fresh symbols. (Thus, $D^\sharp$ has two copies of each function symbol, one of which is marked by $\sharp$.) For the terms $\widetilde{t}_i$ and $\widetilde{t}_j$ that do not contain marked symbols, we write $f_i\,\widetilde{t}_i \rightsquigarrow_D f_j\,\widetilde{t}_j$ if (i) $[\widetilde{t}_i/\widetilde{x}_i][f_1^\sharp/f_1, \dots, f_n^\sharp/f_n]u_i \xrightarrow{\widetilde{\ell}}_{D^\sharp}^* f_j^\sharp\,\widetilde{t}'_j$ and (ii) $\widetilde{t}_j$ is obtained by erasing all the marks in $\widetilde{t}'_j$. We write* **Callseq**(P) *for the set of (possibly infinite) sequences of function symbols:*

$$\{f_1\, g_1\, g_2 \cdots \mid f_1\,\widetilde{s} \rightsquigarrow_D g_1\,\widetilde{t}_1 \rightsquigarrow_D g_2\,\widetilde{t}_2 \rightsquigarrow_D \cdots\}.$$

We write **InfCallseq**(P) *for the subset of* **Callseq**(P) *consisting of infinite sequences, i.e.,* **Callseq**$(P) \cap \{f_1, \dots, f_n\}^\omega$.

For example, for P_{app} above, **Callseq**(P) is the prefix closure of $\{(f_b f_a^5)^\omega\} \cup \{s \cdot \mathsf{app} \mid s$ is a non-empty finite prefix of $(f_b f_a^5)^\omega\}$, and **InfCallseq**(P) is the singleton set $\{(f_b f_a^5)^\omega\}$.

Definition 7 (Call-sequence analysis). *A* priority assignment *for a program P is a function $\Omega : \mathbf{funs}(P) \to \mathbb{N}$ from the set of function symbols of P to the set $\mathbb{N}$ of natural numbers. We write $\models_{csa} (P, \Omega)$ if every infinite call-sequence $g_0 g_1 g_2 \cdots \in$* **InfCallseq**(P) *satisfies the parity condition w.r.t. Ω, i.e., the largest number occurring infinitely often in $\Omega(g_0)\Omega(g_1)\Omega(g_2)\dots$ is even.* Call-sequence analysis *is the problem of, given a program P with a priority assignment Ω, deciding whether $\models_{csa} (P, \Omega)$ holds.*

For example, for P_{app} and the priority assignment $\Omega_{app} = \{\mathsf{app} \mapsto 3, f_a \mapsto 1, f_b \mapsto 2\}$, $\models_{csa} (P_{app}, \Omega_{app})$ holds.

The call-sequence analysis can naturally be reduced to HFL model checking against the trivial LTS $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$ (or validity checking).

Definition 8. *Let $P = (D, t)$ be a program and Ω be a priority assignment for P. The HES $\Phi_{(P,\Omega),csa}$ is $(D^{\dagger_{csa}}, t^{\dagger_{csa}})$, where $D^{\dagger_{csa}}$ and $t^{\dagger_{csa}}$ are defined by:*

$$\{f_1\,\widetilde{x}_1 = t_1, \dots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{csa}} = \left(f_1\;\widetilde{x}_1 =_{\alpha_1} {t_1}^{\dagger_{csa}}; \cdots; f_n\;\widetilde{x}_n =_{\alpha_n} {t_n}^{\dagger_{csa}}\right)$$

$$(\,)^{\dagger_{csa}} = \mathbf{true} \qquad x^{\dagger_{csa}} = x \qquad n^{\dagger_{csa}} = n \qquad (t_1\;\mathsf{op}\;t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}}\;\mathsf{op}\;{t_2}^{\dagger_{csa}}$$

$$(\mathbf{if}\;p(t'_1, \dots, t'_k)\;\mathbf{then}\;t_1\;\mathbf{else}\;t_2)^{\dagger_{csa}} =$$
$$(p({t'_1}^{\dagger_{csa}}, \dots, {t'_k}^{\dagger_{csa}}) \Rightarrow {t_1}^{\dagger_{csa}}) \wedge (\neg p({t'_1}^{\dagger_{csa}}, \dots, {t'_k}^{\dagger_{csa}}) \Rightarrow {t_2}^{\dagger_{csa}})$$

$$(\mathbf{event}\;a; t)^{\dagger_{csa}} = t^{\dagger_{csa}} \qquad (t_1\,t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}}\,{t_2}^{\dagger_{csa}} \qquad (t_1 \Box t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}} \wedge {t_2}^{\dagger_{csa}}.$$

Here, we assume that $\Omega(f_i) \geq \Omega(f_{i+1})$ for each $i \in \{1, \dots, n-1\}$, and $\alpha_i = \nu$ if $\Omega(f_i)$ is even and μ otherwise.

The following theorem states the soundness and completeness of the reduction. See [23] for a proof.

Theorem 4. *Let P be a program and Ω be a priority assignment for P. Then $\models_{csa} (P, \Omega)$ if and only if $\mathtt{L}_0 \models \Phi_{(P,\Omega),csa}$.*

Example 9. For P_{app} and Ω_{app} above, $(P_{app}, \Omega_{app})^{\dagger_{csa}} = (\mathcal{E}, f_b\,5)$, where: $\mathcal{E}$ is:

$$\begin{aligned}&\mathtt{app}\,h\,x =_\mu h\,x; \quad f_b\,x =_\nu (x > 0 \Rightarrow \mathtt{app}\,f_a\,(x-1)) \wedge (x \leq 0 \Rightarrow \mathtt{app}\,f_b\,5);\\ &f_a\,x =_\mu (x > 0 \Rightarrow \mathtt{app}\,f_a\,(x-1)) \wedge (x \leq 0 \Rightarrow \mathtt{app}\,f_b\,5).\end{aligned}$$

Note that $\mathtt{L}_0 \models (P_{app}, \Omega_{app})^{\dagger_{csa}}$ holds.

7.2 From Temporal Verification to Call-Sequence Analysis

This subsection shows a reduction from the temporal verification problem $\mathbf{InfTraces}(P) \cap L \overset{?}{=} \emptyset$ to a call-sequence analysis problem $\overset{?}{\models}_{csa} (P', \Omega)$.

For the sake of simplicity, we assume without loss of generality that every program $P = (D, t)$ in this section is non-terminating and every infinite reduction sequence produces infinite events, so that $\mathbf{FullTraces}(P) = \mathbf{InfTraces}(P)$ holds. We also assume that the ω-regular language L for the temporal verification problem is specified by using a non-deterministic, parity word automaton [10]. We recall the definition of non-deterministic, parity word automata below.

Definition 9 (Parity automaton). *A* non-deterministic parity word automaton *is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, q_I, \Omega)$ where (i) Q is a finite set of states; (ii) Σ is a finite alphabet; (iii) δ, called a transition function, is a* total *map from $Q \times \Sigma$ to 2^Q; (iv) $q_I \in Q$ is the initial state; and (v) $\Omega \in Q \to \mathbb{N}$ is the priority function. A* run *of $\mathcal{A}$ on an ω-word $a_0 a_1 \cdots \in \Sigma^\omega$ is an infinite sequence of states $\rho = \rho(0)\rho(1) \cdots \in Q^\omega$ such that (i) $\rho(0) = q_I$, and (ii) $\rho(i+1) \in \delta(\rho(i), a_i)$ for each $i \in \omega$. An ω-word $w \in \Sigma^\omega$ is* accepted *by $\mathcal{A}$ if, there exists a run ρ of $\mathcal{A}$ on w such that $\mathbf{max}\{\Omega(q) \mid q \in \mathbf{Inf}(\rho)\}$ is even, where $\mathbf{Inf}(\rho)$ is the set of states that occur infinitely often in ρ. We write $\mathcal{L}(\mathcal{A})$ for the set of ω-words accepted by $\mathcal{A}$.*

For technical convenience, we assume below that $\delta(q, a) \neq \emptyset$ for every $q \in Q$ and $a \in \Sigma$; this does not lose generality since if $\delta(q, a) = \emptyset$, we can introduce a new "dead" state q_{dead} (with priority 1) and change $\delta(q, a)$ to $\{q_{dead}\}$. Given a parity automaton $\mathcal{A}$, we refer to each component of $\mathcal{A}$ by $Q_\mathcal{A}$, $\Sigma_\mathcal{A}$, $\delta_\mathcal{A}$, $q_{I,\mathcal{A}}$ and $\Omega_\mathcal{A}$.

Example 10. Consider the automaton $\mathcal{A}_{ab} = (\{q_a, q_b\}, \{\mathtt{a}, \mathtt{b}\}, \underline{\delta, q_a, \Omega})$, where δ is as given in Fig. 4, $\Omega(q_a) = 0$, and $\Omega(q_b) = 1$. Then, $\mathcal{L}(\mathcal{A}_{ab}) = \overline{(\mathtt{a}^*\mathtt{b})^\omega} = (\mathtt{a}^*\mathtt{b})^*\mathtt{a}^\omega$.

The goal of this subsection is, given a program P and a parity word automaton $\mathcal{A}$, to construct another program P' and a priority assignment Ω for P', such that $\mathbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ if and only if $\models_{csa} (P', \Omega)$.

Note that a necessary and sufficient condition for $\mathbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ is that no trace in $\mathbf{InfTraces}(P)$ has a run whose priority sequence satisfies the parity condition; in other words, for every sequence in $\mathbf{InfTraces}(P)$, and for every run for the sequence, the largest priority that occurs in the associated priority sequence is odd. As explained at the beginning of this section, we reduce this condition to a call sequence analysis problem by appropriately duplicating functions in a given program. For example, recall the program P_0:

$$(\{f = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f)\}, f)\,.$$

It is translated to P_0':

$$\begin{aligned}(\{f_b &= \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f_b),\\ f_a &= \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f_b)\},\ f_b),\end{aligned}$$

where c is some (closed) boolean expression. Since the largest priorities encountered before calling f_a and f_b (since the last recursive call) respectively are 0 and 1, we assign those priorities plus 1 (to flip odd/even-ness) to f_a and f_b respectively. Then, the problem of $\mathbf{InfTraces}(P_0) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ is reduced to $\models_{csa} (P_0', \{f_a \mapsto 1, f_b \mapsto 2\})$. Note here that the priorities of f_a and f_b represent *summaries* of the priorities (plus one) that occur in the run of the automaton until f_a and f_b are respectively called since the last recursive call; thus, the largest priority of states that occur infinitely often in the run for an infinite trace is equivalent to the largest priority that occurs infinitely often in the sequence of summaries $(\Omega(f_1)-1)(\Omega(f_2)-1)(\Omega(f_3)-1)\cdots$ computed from a corresponding call sequence $f_1 f_2 f_3 \cdots$.

Due to the presence of higher-order functions, the general reduction is more complicated than the example above. First, we need to replicate not only function symbols, but also arguments. For example, consider the following variation P_1 of P_0 above:

$$(\{g\,k = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; k)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; k), \quad f = g\,f\}, \quad f)\,.$$

Here, we have just made the calls to f indirect, by preparing the function g. Obviously, the two calls to k in the body of g must be distinguished from each other, since different priorities are encountered before the calls. Thus, we duplicate the argument k, and obtain the following program P_1':

$$(\{g\,k_a\,k_b = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; k_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; k_b), f_a = g\,f_a\,f_b, f_b = g\,f_a\,f_b\}, f_a).$$

Then, for the priority assignment $\Omega = \{f_a \mapsto 1, f_b \mapsto 2, g \mapsto 1\}$, $\mathbf{InfTraces}(P_1) \cap \mathcal{L}(\mathcal{A}_{ab}) = \emptyset$ if and only if $\models_{csa} (P_1', \Omega)$. Secondly, we need to take into account not only the priorities of states visited by $\mathcal{A}$, but also the states themselves. For example, if we have a function definition $f\,h = h(\mathbf{event}\ \mathtt{a}; f\,h)$, the largest priority encountered before f is recursively called in the body of f depends on the priorities encountered inside h, *and also* the state of $\mathcal{A}$ when h uses the argument **event** a; f (because the state after the a event depends on the previous state in general). We, therefore, use *intersection types* (a la Kobayashi and Ong's intersection types for HORS model checking [21]) to represent summary information on how each function traverses states of the automaton, and replicate each function and its arguments for each type. We thus formalize the translation as an intersection-type-based program transformation; related transformation techniques are found in [8,11,12,20,38].

Definition 10. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \Omega)$ be a non-deterministic parity word automaton. Let q and m range over Q and the set codom(Ω) of priorities respectively. The set* $\mathbf{Types}_\mathcal{A}$ *of* intersection types*, ranged over by θ, is defined by:*

$$\theta ::= q \mid \rho \to \theta \qquad\qquad \rho ::= \mathtt{int} \mid \textstyle\bigwedge_{1\le i\le k}(\theta_i, m_i)$$

We assume a certain total order $<$ on $\mathbf{Types}_\mathcal{A} \times \mathbb{N}$, and require that in $\bigwedge_{1\le i\le k}(\theta_i, m_i)$, $(\theta_i, m_i) < (\theta_j, m_j)$ holds for each $i < j$.

We often write $(\theta_1, m_1) \wedge \cdots \wedge (\theta_k, m_k)$ for $\bigwedge_{1\le i\le k}(\theta_i, m_i)$, and $\top$ when $k = 0$. Intuitively, the type q describes expressions of simple type $\star$, which may be evaluated when the automaton $\mathcal{A}$ is in the state q (here, we have in mind an execution of the *product* of a program and the automaton, where the latter takes events produced by the program and changes its states). The type $(\bigwedge_{1\le i\le k}(\theta_i, m_i)) \to \theta$ describes functions that take an argument, use it according to types $\theta_1, \dots, \theta_k$, and return a value of type θ. Furthermore, the part m_i describes that the argument may be used as a value of type θ_i only when the largest priority visited since the function is called is m_i. For example, given the automaton in Example 10, the function $\lambda x.$(**event** a; x) may have types $(q_a, 0) \to q_a$ and $(q_a, 0) \to q_b$, because the body may be executed from state q_a or q_b (thus, the return type may be any of them), but x is used only when the automaton is in state q_a and the largest priority visited is 1. In contrast, $\lambda x.$(**event** b; x) have types $(q_b, 1) \to q_a$ and $(q_b, 1) \to q_b$.

Using the intersection types above, we shall define a type-based transformation relation of the form $\Gamma \vdash_\mathcal{A} t : \theta \Rightarrow t'$, where t and t' are the source and target terms of the transformation, and Γ, called an *intersection type environment*, is a finite set of type bindings of the form $x : \mathtt{int}$ or $x : (\theta, m, m')$. We allow multiple type bindings for a variable x except for $x : \mathtt{int}$ (i.e. if $x : \mathtt{int} \in \Gamma$, then this must be the unique type binding for x in Γ). The binding $x : (\theta, m, m')$ means that x should be used as a value of type θ when the largest priority visited is m; m' is auxiliary information used to record the largest priority encountered so far.

The transformation relation $\Gamma \vdash_\mathcal{A} t : \theta \Rightarrow t'$ is inductively defined by the rules in Fig. 5. (For technical convenience, we have extended terms with λ-abstractions; they may occur only at top-level function definitions.) In the figure, $[k]$ denotes the set $\{i \in \mathbb{N} \mid 1 \le i \le k\}$. The operation $\Gamma \uparrow m$ used in the figure is defined by:

$$\Gamma \uparrow m = \{x : \mathtt{int} \mid x : \mathtt{int} \in \Gamma\} \cup \{x : (\theta, m_1, \mathbf{max}(m_2, m)) \mid x : (\theta, m_1, m_2) \in \Gamma\}$$

The operation is applied when the priority m is encountered, in which case the largest priority encountered is updated accordingly. The key rules are IT-Var, IT-Event, IT-App, and IT-Abs. In IT-Var, the variable x is replicated for each type; in the target of the translation, $x_{\theta,m}$ and $x_{\theta',m'}$ are treated as different variables if $(\theta, m) \neq (\theta', m')$. The rule IT-Event reflects the state change caused by the event a to the type and the type environment. Since the state change may be non-deterministic, we transform t for each of the next states $q_1, \ldots, q_n$, and combine the resulting terms with non-deterministic choice. The rule IT-App and IT-Abs replicates function arguments for each type. In addition, in IT-App, the operation $\Gamma \uparrow m_i$ reflects the fact that t_2 is used as a value of type θ_i after the priority m_i is encountered. The other rules just transform terms in a compositional manner. If target terms are ignored, the entire rules are close to those of Kobayashi and Ong's type system for HORS model checking [21].

$$\frac{}{\Gamma \vdash_{\mathcal{A}} () : q \Rightarrow ()} \quad \text{(IT-Unit)}$$

$$\frac{}{\Gamma, x : \mathtt{int} \vdash_{\mathcal{A}} x : \mathtt{int} \Rightarrow x_{\mathtt{int}}} \quad \text{(IT-VarInt)}$$

$$\frac{}{\Gamma, x : (\theta, m, m) \vdash_{\mathcal{A}} x : \theta \Rightarrow x_{\theta,m}} \quad \text{(IT-Var)}$$

$$\frac{}{\Gamma \vdash_{\mathcal{A}} n : \mathtt{int} \Rightarrow n} \quad \text{(IT-Int)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : \mathtt{int} \Rightarrow t_1' \qquad \Gamma \vdash_{\mathcal{A}} t_2 : \mathtt{int} \Rightarrow t_2'}{\Gamma \vdash_{\mathcal{A}} t_1 \,\mathtt{op}\, t_2 : \mathtt{int} \Rightarrow t_1' \,\mathtt{op}\, t_2'} \quad \text{(IT-Op)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} t_i : \mathtt{int} \Rightarrow t_i' \quad (\text{for each } i \in [k]) \\ \Gamma \vdash_{\mathcal{A}} t_{k+1} : q \Rightarrow t_{k+1}' \\ \Gamma \vdash_{\mathcal{A}} t_{k+2} : q \Rightarrow t_{k+2}' \\ \widetilde{t} = t_1, \ldots, t_k \qquad \widetilde{t'} = t_1', \ldots, t_k'\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} \mathbf{if}\ p(\widetilde{t})\ \mathbf{then}\ t_{k+1}\ \mathbf{else}\ t_{k+2} : q \\ \Rightarrow \mathbf{if}\ p(\widetilde{t'})\ \mathbf{then}\ t_{k+1}'\ \mathbf{else}\ t_{k+2}'\end{array}} \quad \text{(IT-If)}$$

$$\frac{\begin{array}{c}\delta_A(q, a) = \{q_1, \ldots, q_k\} \\ \Gamma \uparrow \Omega_A(q_i) \vdash_{\mathcal{A}} t : q_i \Rightarrow t_i' \quad (\text{for each } i \in [k])\end{array}}{\Gamma \vdash_{\mathcal{A}} (\mathbf{event}\ a; t) : q \Rightarrow (\mathbf{event}\ a; t_1' \Box \cdots \Box t_k')} \quad \text{(IT-Event)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : q \Rightarrow t_1' \qquad \Gamma \vdash_{\mathcal{A}} t_2 : q \Rightarrow t_2'}{\Gamma \vdash_{\mathcal{A}} t_1 \Box t_2 : q \Rightarrow t_1' \Box t_2'} \quad \text{(IT-NonDet)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} t_1 : \mathtt{int} \to \theta \Rightarrow t_1' \\ \Gamma \vdash_{\mathcal{A}} t_2 : \mathtt{int} \Rightarrow t_2'\end{array}}{\Gamma \vdash_{\mathcal{A}} t_1\ t_2 : \theta \Rightarrow t_1'\ t_2'} \quad \text{(IT-AppInt)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} t_1 : \bigwedge_{1 \le i \le k} (\theta_i, m_i) \to \theta \Rightarrow t_1' \\ \Gamma \uparrow m_i \vdash_{\mathcal{A}} t_2 : \theta_i \Rightarrow t_{2,i}' \ (\text{for each } i \in [k])\end{array}}{\Gamma \vdash_{\mathcal{A}} t_1\ t_2 : \theta \Rightarrow t_1'\ t_{2,1}' \ \ldots\ t_{2,k}'} \quad \text{(IT-App)}$$

$$\frac{\Gamma, x : \mathtt{int} \vdash_{\mathcal{A}} t : \theta \Rightarrow t' \qquad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : \mathtt{int} \to \theta \Rightarrow \lambda x_{\mathtt{int}}.t'} \quad \text{(IT-AbsInt)}$$

$$\frac{\begin{array}{c}\Gamma \cup \{x : (\theta_i, m_i, 0) \mid i \in [k]\} \vdash_{\mathcal{A}} t : \theta' \Rightarrow t' \\ x \notin dom(\Gamma)\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} \lambda x.t : \bigwedge_{1 \le i \le k} (\theta_i, m_i) \to \theta' \\ \Rightarrow \lambda x_{\theta_1,m_1} \ldots x_{\theta_k,m_k}.t'\end{array}} \quad \text{(IT-Abs)}$$

Fig. 5. Type-based transformation rules for terms

We now define the transformation for programs. A *top-level type environment* Ξ is a finite set of type bindings of the form $x : (\theta, m)$. Like intersection type environments, Ξ may have more than one binding for each variable. We write $\Xi \vdash_{\mathcal{A}} t : \theta$ to mean $\{x : (\theta, m, 0) \mid x : (\theta, m) \in \Xi\} \vdash_{\mathcal{A}} t : \theta$. For a set D of function definitions, we write $\Xi \vdash_{\mathcal{A}} D \Rightarrow D'$ if $dom(D') = \{ f_{\theta,m} \mid f : (\theta, m) \in \Xi \}$ and $\Xi \vdash_{\mathcal{A}} D(f) : \theta \Rightarrow D'(f_{\theta,m})$ for every $f : (\theta, m) \in \Xi$. For a program $P = (D, t)$, we write $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ if $P' = (D', t')$, $\Xi \vdash_{\mathcal{A}} D \Rightarrow D'$ and $\Xi \vdash_{\mathcal{A}} t : q_I \Rightarrow t'$, with $\Omega'(f_{\theta,m}) = m+1$ for each $f_{\theta,m} \in \mathit{dom}(D')$. We just write $\vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ if $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ holds for some Ξ.

Example 11. Consider the automaton $\mathcal{A}_{ab}$ in Example 10, and the program $P_2 = (D_2, f\,5)$ where D_2 consists of the following function definitions:

$$\begin{aligned}&g\;k = (\textbf{event}\;\texttt{a}; k)\Box(\textbf{event}\;\texttt{b}; k),\\ &f\;x = \textbf{if}\;x > 0\;\textbf{then}\;g\,(f(x-1))\;\textbf{else}\;(\textbf{event}\;\texttt{b}; f\,5).\end{aligned}$$

Let Ξ be: $\{g : ((q_a, 0) \wedge (q_b, 1) \to q_a, 0), g : ((q_a, 0) \wedge (q_b, 1) \to q_b, 0), f : (\texttt{int} \to q_a, 0), f : (\texttt{int} \to q_b, 1)\}$. Then, $\Xi \vdash_{\mathcal{A}} P_1 \Rightarrow ((D_2', f_{\texttt{int}\to q_a,0}\,5), \Omega)$ where:

$$\begin{aligned}
D_2' = \{&g_{(q_a,0)\wedge(q_b,1)\to q_a,0}\;k_{q_a,0}\;k_{q_b,1} = t_g, \quad g_{(q_a,0)\wedge(q_b,1)\to q_b,0}\;k_{q_a,0}\;k_{q_b,1} = t_g,\\
&f_{\texttt{int}\to q_a,0}\;x_{\texttt{int}} = t_{f,q_a}, \quad f_{\texttt{int}\to q_b,1}\;x_{\texttt{int}} = t_{f,q_b}\}\\
t_g = {}&(\textbf{event}\;\texttt{a}; k_{q_a,0})\Box(\textbf{event}\;\texttt{b}; k_{q_b,1}),\\
t_{f,q} = {}&\textbf{if}\;x_{\texttt{int}} > 0\;\textbf{then}\\
&\qquad g_{(q_a,0)\wedge(q_b,1)\to q,0}\,(f_{\texttt{int}\to q_a,0}(x_{\texttt{int}}-1))\,(f_{\texttt{int}\to q_b,1}(x_{\texttt{int}}-1))\\
&\quad\textbf{else}\;(\textbf{event}\;\texttt{b}; f_{\texttt{int}\to q_b,1}\,5), \qquad\qquad (\text{for each } q \in \{q_a, q_b\})\\
\Omega = \{&g_{(q_a,0)\wedge(q_b,1)\to q_a,0} \mapsto 1, g_{(q_a,0)\wedge(q_b,1)\to q_b,0} \mapsto 1, f_{\texttt{int}\to q_a,0} \mapsto 1, f_{\texttt{int}\to q_b,1} \mapsto 2\}.
\end{aligned}$$

Notice that f, g, and the arguments of g have been duplicated. Furthermore, whenever $f_{\theta,m}$ is called, the largest priority that has been encountered since the last recursive call is m. For example, in the then-clause of $f_{\texttt{int}\to q_a,0}$, $f_{\texttt{int}\to q_b,1}(x-1)$ may be called through $g_{(q_a,0)\wedge(q_b,1)\to q_a,0}$. Since $g_{(q_a,0)\wedge(q_b,1)\to q_a,0}$ uses the second argument only after an event $\texttt{b}$, the largest priority encountered is 1. This property is important for the correctness of our reduction.

The following theorems below claim that our reduction is sound and complete, and that there is an effective algorithm for the reduction: see [23] for proofs.

Theorem 5. *Let P be a program and $\mathcal{A}$ be a parity automaton. Suppose that $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega)$. Then* $\textbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ *if and only if* $\models_{csa} (P', \Omega)$.

Theorem 6. *For every P and $\mathcal{A}$, one can effectively construct Ξ, P' and Ω such that $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega)$.*

The proof of Theorem 6 above also implies that the reduction from temporal property verification to call-sequence analysis can be performed in polynomial time. Combined with the reduction from call-sequence analysis to HFL model checking, we have thus obtained a polynomial-time reduction from the temporal verification problem $\textbf{InfTraces}(P) \overset{?}{\subseteq} \mathcal{L}(\mathcal{A})$ to HFL model checking.

8 Related Work

As mentioned in Sect. 1, our reduction from program verification problems to HFL model checking problems has been partially inspired by the translation of

Kobayashi et al. [19] from HORS model checking to HFL model checking. As in their translation (and unlike in previous applications of HFL model checking [28, 42]), our translation switches the roles of properties and models (or programs) to be verified. Although a combination of their translation with Kobayashi's reduction from program verification to HORS model checking [17,18] yields an (indirect) translation from *finite-data* programs to pure HFL model checking problems, the combination does not work for infinite-data programs. In contrast, our translation is sound and complete even for infinite-data programs. Among the translations in Sects. 5, 6 and 7, the translation in Sect. 7.2 shares some similarity to their translation, in that functions and their arguments are replicated for each priority. The actual translations are however quite different; ours is type-directed and optimized for a given automaton, whereas their translation is not. This difference comes from the difference of the goals: the goal of [19] was to clarify the relationship between HORS and HFL, hence their translation was designed to be independent of an automaton. The proof of the correctness of our translation in Sect. 7 is much more involved due to the need for dealing with integers. Whilst the proof of [19] could reuse the type-based characterization of HORS model checking [21], we had to generalize arguments in both [19,21] to work on infinite-data programs.

Lange et al. [28] have shown that various process equivalence checking problems (such as bisimulation and trace equivalence) can be reduced to (pure) HFL model checking problems. The idea of their reduction is quite different from ours. They reduce processes to LTSs, whereas we reduce programs to HFL formulas.

Major approaches to automated or semi-automated higher-order program verification have been HORS model checking [17,18,22,27,31,33,43], (refinement) type systems [14,24,34–36,39,41,44], Horn clause solving [2,7], and their combinations. As already discussed in Sect. 1, compared with the HORS model checking approach, our new approach provides more uniform, streamlined methods. Whilst the HORS model checking approach is for fully automated verification, our approach enables various degrees of automation: after verification problems are automatically translated to $\mathrm{HFL}_{\mathbf{Z}}$ formulas, one can prove them (i) interactively using a proof assistant like Coq (see [23]), (ii) semi-automatically, by letting users provide hints for induction/co-induction and discharging the rest of proof obligations by (some extension of) an SMT solver, or (iii) fully automatically by recasting the techniques used in the HORS-based approach; for example, to deal with the ν-only fragment of $\mathrm{HFL}_{\mathbf{Z}}$, we can reuse the technique of predicate abstraction [22]. For a more technical comparison between the HORS-based approach and our HFL-based approach, see [23].

As for type-based approaches [14,24,34–36,39,41,44], most of the refinement type systems are (i) restricted to safety properties, and/or (ii) incomplete. A notable exception is the recent work of Unno et al. [40], which provides a relatively complete type system for the classes of properties discussed in Sect. 5. Our approach deals with a wider class of properties (cf. Sects. 6 and 7). Their "relative completeness" property relies on Godel coding of functions, which cannot be exploited in practice.

The reductions from program verification to Horn clause solving have recently been advocated [2–4] or used [34,39] (via refinement type inference problems) by a number of researchers. Since Horn clauses can be expressed in a fragment of HFL without modal operators, fixpoint alternations (between ν and μ), and higher-order predicates, our reductions to HFL model checking may be viewed as extensions of those approaches. Higher-order predicates and fixpoints over them allowed us to provide sound and complete characterizations of properties of higher-order programs for a wider class of properties. Bjørner et al. [4] proposed an alternative approach to obtaining a complete characterization of safety properties, which defunctionalizes higher-order programs by using algebraic data types and then reduces the problems to (first-order) Horn clauses. A disadvantage of that approach is that control flow information of higher-order programs is also encoded into algebraic data types; hence even for finite-data higher-order programs, the Horn clauses obtained by the reduction belong to an undecidable fragment. In contrast, our reductions yield pure HFL model checking problems for finite-data programs. Burn et al. [7] have recently advocated the use of *higher-order* (constrained) Horn clauses for verification of safety properties (i.e., which correspond to the negation of may-reachability properties discussed in Sect. 5.1 of the present paper) of higher-order programs. They interpret recursion using the least fixpoint semantics, so their higher-order Horn clauses roughly corresponds to a fragment of the $\text{HFL}_{\mathbf{Z}}$ without modal operators and fixpoint alternations. They have not shown a general, concrete reduction from safety property verification to higher-order Horn clause solving.

The characterization of the reachability problems in Sect. 5 in terms of formulas without modal operators is a reminiscent of predicate transformers [9,13] used for computing the weakest preconditions of imperative programs. In particular, [5] and [13] respectively used least fixpoints to express weakest preconditions for while-loops and recursions.

9 Conclusion

We have shown that various verification problems for higher-order functional programs can be naturally reduced to (extended) HFL model checking problems. In all the reductions, a program is mapped to an HFL formula expressing the property that the behavior of the program is correct. For developing verification tools for higher-order functional programs, our reductions allow us to focus on the development of (automated or semi-automated) $\text{HFL}_{\mathbf{Z}}$ model checking tools (or, even more simply, theorem provers for $\text{HFL}_{\mathbf{Z}}$ without modal operators, as the reductions of Sects. 5 and 7 yield HFL formulas without modal operators). To this end, we have developed a prototype model checker for pure HFL (without integers), which will be reported in a separate paper. Work is under way to develop $\text{HFL}_{\mathbf{Z}}$ model checkers by recasting the techniques [22,26,27,43] developed for the HORS-based approach, which, together with the reductions presented in this paper, would yield fully automated verification tools. We have also started building a Coq library for interactively proving $\text{HFL}_{\mathbf{Z}}$ formulas, as briefly discussed in [23]. As a final remark, although one may fear that our reductions may map program verification problems to "harder" problems due to the expressive power of $\mathrm{HFL}_{\mathbf{Z}}$, it is actually not the case at least for the classes of problems in Sects. 5 and 6, which use the only alternation-free fragment of $\mathrm{HFL}_{\mathbf{Z}}$. The model checking problems for μ-only or ν-only $\mathrm{HFL}_{\mathbf{Z}}$ are semi-decidable and co-semi-decidable respectively, like the source verification problems of may/must-reachability and their negations of closed programs.

Acknowledgment. We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP16K16004.

References

1. Axelsson, R., Lange, M., Somla, R.: The complexity of model checking higher-order fixpoint logic. Logical Methods Comput. Sci. **3**(2), 1–33 (2007)
2. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
3. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT 2012, EPiC Series in Computing, vol. 20, pp. 3–11. EasyChair (2012)
4. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Higher-order program verification as satisfiability modulo theories with algebraic data-types. CoRR, abs/1306.5264 (2013)
5. Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Börger, E. (ed.) Computation Theory and Logic. LNCS, vol. 270, pp. 20–36. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18170-9_151
6. Blume, M., Acar, U.A., Chae, W.: Exception handlers as extensible cases. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 273–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_20
7. Burn, T.C., Ong, C.L., Ramsay, S.J.: Higher-order constrained horn clauses for verification. PACMPL **2**(POPL), 11:1–11:28 (2018)
8. Carayol, A., Serre, O.: Collapsible pushdown automata and labeled recursion schemes: equivalence, safety and effective selection. In: LICS 2012, pp. 165–174. IEEE (2012)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
10. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
11. Grellois, C., Melliès, P.: Relational semantics of linear logic and higher-order model checking. In: Proceedings of CSL 2015, LIPIcs, vol. 41, pp. 260–276 (2015)
12. Haddad, A.: Model checking and functional program transformations. In: Proceedings of FSTTCS 2013, LIPIcs, vol. 24, pp. 115–126 (2013)
13. Hesselink, W.H.: Predicate-transformer semantics of general recursion. Acta Inf. **26**(4), 309–332 (1989)

14. Hofmann, M., Chen, W.: Abstract interpretation from Büchi automata. In: Proceedings of CSL-LICS 2014, pp. 51:1–51:10. ACM (2014)
15. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Prog. Lang. Syst. **27**(2), 264–313 (2005)
16. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_15
17. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of POPL, pp. 416–428. ACM Press (2009)
18. Kobayashi, N.: Model checking higher-order programs. J. ACM **60**(3), 1–62 (2013)
19. Kobayashi, N., Lozes, É., Bruse, F.: On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In: Proceedings of POPL 2017, pp. 246–259 (2017)
20. Kobayashi, N., Matsuda, K., Shinohara, A., Yaguchi, K.: Functional programs as compressed data. High.-Order Symbolic Comput. **25**(1), 39–84 (2013)
21. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009, pp. 179–188 (2009)
22. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI, pp. 222–233. ACM Press (2011)
23. Kobayashi, N., Tsukada, T., Watanabe, K.: Higher-order program verification via HFL model checking. CoRR abs/1710.08614 (2017). http://arxiv.org/abs/1710.08614
24. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: Proceedings of CSL-LICS 2014, pp. 59:1–59:10. ACM (2014)
25. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
26. Kuwahara, T., Sato, R., Unno, H., Kobayashi, N.: Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 287–303. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_17
27. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 392–411. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_21
28. Lange, M., Lozes, É., Guzmán, M.V.: Model-checking process equivalences. Theor. Comput. Sci. **560**, 326–347 (2014)
29. Ledesma-Garza, R., Rybalchenko, A.: Binary reachability analysis of higher order functional programs. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 388–404. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_26
30. Lozes, É.: A type-directed negation elimination. In: Proceedings FICS 2015, EPTCS, vol. 191, pp. 132–142 (2015)
31. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Proceedings of POPL 2016, pp. 57–68 (2016)
32. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press (2006)
33. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL, pp. 587–598. ACM Press (2011)

34. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. PLDI **2008**, 159–169 (2008)
35. Skalka, C., Smith, S.F., Horn, D.V.: Types and trace effects of higher order programs. J. Funct. Program. **18**(2), 179–249 (2008)
36. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of POPL, pp. 119–130. ACM (2010)
37. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 275–289. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29822-6_22
38. Tsukada, T., Ong, C.L.: Compositional higher-order model checking via ω-regular games over Böhm trees. In: Proceedings of CSL-LICS 2014, pp. 78:1–78:10. ACM (2014)
39. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP 2009, pp. 277–288. ACM (2009)
40. Unno, H., Satake, Y., Terauchi, T.: Relatively complete refinement type system for verification of higher-order non-deterministic programs. PACMPL **2**(POPL), 12:01–12:29 (2018)
41. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: POPL 2013. pp. 75–86. ACM (2013)
42. Viswanathan, M., Viswanathan, R.: A higher order modal fixed point logic. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 512–528. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_33
43. Watanabe, K., Sato, R., Tsukada, T., Kobayashi, N.: Automatically disproving fair termination of higher-order functional programs. In: Proceedings of ICFP 2016, pp. 243–255. ACM (2016)
44. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of ICFP 2015, pp. 400–411. ACM (2015)