Reduction from Branching-Time Property Verification of Higher-Order Programs to HFL Validity Checking

Keiichi Watanabe*
Department of Computer Science
University of Tokyo
Japan
udon.watanabe@gmail.com

Takeshi Tsukada
Department of Computer Science
University of Tokyo
Japan
tsukada@is.s.u-tokyo.ac.jp

Hiroki Oshikawa
Department of Computer Science
University of Tokyo
Japan
oshi@is.s.u-tokyo.ac.jp

Naoki Kobayashi
Department of Computer Science
University of Tokyo
Japan
koba@is.s.u-tokyo.ac.jp

Abstract

Various methods have recently been proposed for temporal property verification of higher-order programs. In those methods, however, either temporal properties were limited to *linear-time* ones, or target programs were limited to *finite-data* programs. In this paper, we extend Kobayashi et al.'s recent method for verification of linear-time temporal properties based on HFL_Z model checking, to deal with branching-time properties. We formalize branching-time property verification problems as an extension of HORS model checking called HORS_Z model checking, present a sound and complete reduction to validity checking of (modal-free) HFL_Z formulas, and prove its correctness. The correctness of the reduction subsumes the decidability of HORS model checking. The HFL_Z formula obtained by the reduction from a HORS_Z model checking problem can be considered a kind of verification condition for the orignal model checking problem. We also discuss interactive and automated methods for discharging the verification condition.

CCS Concepts • **Theory of computation → Logic and verification**;

Keywords higher-order program verification, higher-order fixpoint logic

ACM Reference Format:
Keiichi Watanabe, Takeshi Tsukada, Hiroki Oshikawa, and Naoki Kobayashi. 2019. Reduction from Branching-Time Property Verification of Higher-Order Programs to HFL Validity Checking. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3294032.3294077

*now at Google

PEPM '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6226-9/19/01…$15.00
https://doi.org/10.1145/3294032.3294077

1 Introduction

Verification methods for temporal properties of higher-order functional programs have been actively studied recently [12, 14, 18, 22, 23, 26, 28, 29, 35]. The existing methods, however, have limitations: either (i) verifiable properties are restricted to *linear-time* temporal properties (like "there exists an execution path such that ..." or "every execution path satisfies ...", not branching-time properties like "after any finite execution sequence such that ..., there exists a path such that ..." or "there exists a path that reaches a state from which every execution satisfies ..."[1]), or (ii) target programs are restricted to *finite-data* higher-order programs (i.e., higher-order functional programs where base type values are restricted to those from finite data domains such as booleans, not those from infinite data domains such as integers, lists, and trees). For example, Kobayashi [17, 18] and Lester et al. [26] have shown that certain temporal property verification problems can be reduced to HORS model checking, but target programs are limited to finite-data ones. (Kobayashi et al. [21] have shown how to use predicate abstraction to deal with infinite-data programs, but the method works only for safety properties, not liveness properties.). The technique of Murase et al. [28] deals with infinite-data programs,

[1] $\mathrm{AG}(\mathrm{E}\varphi)$ and $\mathrm{EF}(\mathrm{A}\varphi)$ in CTL*

but it relies on Vardi's reduction [32] from linear-time temporal property verification to fair termination, hence restricted to verification of linear-time properties.

The aim of the present work is, therefore, to establish a foundation for *branching-time* property verification of *infinite-data* higher-order programs. We first introduce HORS_Z, an extension of higher-order recursion schemes (HORS) [15, 30] with integers, and formalize verification problems for *branching-time* properties of *infinite-data* higher-order programs as HORS_Z model checking problems. We then extend the recent result of Kobayashi et al. [22]. They have shown that linear-time temporal verification problems for infinite-data programs can be reduced to the validity checking of modal-free HFL_Z formulas, where HFL_Z is an extension of higher-order modal fixpoint logic (HFL) [33] with integers. We extend their reduction to obtain a sound and complete reduction from HFL_Z model checking to modal-free HFL_Z validity checking, and prove its correctness.

The HFL_Z formula φ obtained by our reduction from a HORS_Z model checking problem can be considered a kind of "verification condition" á la that for Hoare logic, in that φ is valid if and only if the answer to the source (model checking) problem is "yes". The differences from the usual verification conditions based on Hoare logic are: (1) our "verification condition" characterize *temporal properties* of *higher-order* programs, instead of partial/total correctness of imperative programs, (2) our verification condition is a formula of *higher-order fixpoint* logic, and (3) loop invariant annotations are not required (thanks to the presence of fixpoint operators; instead, it becomes more challenging to discharge our verification conditions). Our approach of reducing verification problems to validity checking in a fixpoint logic is also related to the recent popular approach of reductions from program verification problems to CHC (constrained Horn clauses) solving [3]. In fact, HFL_Z may be considered an extension of CHC with higher-order predicates and fixpoint alternation.

In the present paper, we also discuss how to prove the validity of HFL_Z formulas obtained by our reduction, in an interactive manner by using Coq proof assistant. Thus, combined with the reduction from HORS_Z verification, our technique yields a semi-automated verification method for branching-time properties of higher-order programs (where the first phase is fully automatic and the second one requires human interactions). We show that HFL_Z can be formalized quite naturally in Coq, and inductive and co-inductive reasoning (which is indispensable for temporal property verification) can also be applied in a natural manner. This is in a sharp contrast to an approach of directly formalizing and solving HORS_Z model checking problems in Coq; if we were to formalize HORS_Z model checking in Coq, we would have to formalize complex concepts such as infinite trees, alternating parity tree automata, etc. We have also developed Coq tactics to reduce the burden of proving HFL_Z formulas interactively.

A fully automated prover for HFL_Z formulas (that would, combined with our reduction, yield a fully automated verification tool) is also under development, but in the present paper, we sketch the approach only briefly.

The contributions of the paper are summarized as follows.

1. The formulation of verification problems for temporal properties of higher-order programs as HORS_Z model checking problems.
2. The reduction from HORS_Z model checking problems to HFL_Z validity checking problems and a proof of its correctness. The proof is highly non-trivial; in fact, since our reduction yields a pure HFL validity checking problem (which is obviously decidable) when applied to a pure HORS model checking problem, the correctness of our reduction subsumes the decidability of (pure) HORS model checking (which had been considered difficult and open for several years until Ong [30] solved it).
3. The development of a Coq library and tactics for proving HFL_Z formulas.

The rest of this paper is structured as follows. Section 2 defines the source and target problems, i.e. HORS_Z model checking and modal-free HFL_Z validity checking problems. Section 3 defines our reduction and sketches a proof of its soundness and completeness; the concrete proof is found in [36]. Section 4 describes the Coq library and tactics for solving HFL_Z validity checking problems in Coq, and also briefly sketches an approach to solving HFL_Z validity checking problems automatically. Section 5 discusses related work, and Section 6 concludes the paper. Omitted definitions and proofs are given in the longer version of the paper [36].

2 Source/Target Problems

In this section, we introduce the source and target problems of our reduction, i.e., the HORS_Z model checking problem and the (modal-free) HFL_Z validity checking problem. We write $\mathbf{Z}$ for the set of integers, ranged over by n; we also write $\mathbf{Z}_+$ for the set of positive integers, and $\mathbf{N}$ for the set of natural numbers (i.e. non-negative integers). We assume a set **Pred** of primitive (binary) integer predicates (such as $<$), ranged over by p, and a set **Op** of integer operations (such as $+$), ranged over by op. We assume that if $p \in \mathbf{Pred}$, then its negation, written $\neg p$, also belongs to **Pred**; when we write $\neg p(t_1, t_2)$ it should be parsed as $(\neg p)(t_1, t_2)$ but it can intuitively be read as $\neg(p(t_1, t_2))$. For a map f, we write $dom(f)$ and $codom(f)$ for the domain and codomain of f respectively. We often write $\widetilde{\cdot}$ for a sequence, and $|\cdot|$ for the length of the sequence. For example, we write $\widetilde{x}$ for a sequence of variables $x_1, \ldots, x_k$.

2.1 HORS$_\mathsf{Z}$ Model Checking

We introduce HORS$_\mathsf{Z}$, an extension of higher-order recursion schemes (HORS) [15, 30] with integers, and define the model checking problem. A HORS$_\mathsf{Z}$ can be considered a simply-typed, call-by-name higher-order functional program (with recursion) that generates a possibly infinite tree. Using Kobayashi's idea [17, 18], temporal verification problems for higher-order functional programs can be easily converted to HORS$_\mathsf{Z}$ model checking problems, by transforming a higher-order functional program to a HORS$_\mathsf{Z}$ that generates a "computation tree", which represents all the possible executions of the program.

2.1.1 Infinite Trees and Tree Automata

We recall here the standard notion of (labeled, possibly infinite) trees and tree automata [13].

A *ranked alphabet* is a map from a finite set of symbols to non-negative integers. For a ranked alphabet Σ and a symbol $\mathsf{a} \in \mathit{dom}(\Sigma)$, $\Sigma(\mathsf{a})$ is called the *arity* of a. A *tree* is a subset S of $\mathbf{Z}_+^*$ such that $\pi i \in S$ implies $\{\pi, \pi 1, \pi 2, \ldots, \pi(i-1)\} \subseteq S$. Let $\mathcal{L}$ be a set of symbols. An *$\mathcal{L}$-labeled tree* is a function from a tree to $\mathcal{L}$. A $\mathit{dom}(\Sigma)$-labeled tree T is called a *Σ-labeled ranked tree* when $T(\pi) = \mathsf{a}$ implies $\{i \mid \pi i \in \mathit{dom}(T)\} = \{1, \ldots, \Sigma(\mathsf{a})\}$.

If $T_1, \ldots, T_n$ are labeled trees and $\mathsf{a} \in \mathcal{L}$, we often write $\mathsf{a}(T_1, \ldots, T_n)$ for a labeled tree T such that (i) $\mathit{dom}(T) = \{\epsilon\} \cup \{i\pi \mid i \in \{1, \ldots, n\}, \pi \in \mathit{dom}(T_i)\}$, (ii) $T(\epsilon) = \mathsf{a}$, and (iii) $T(i\pi) = T_i(\pi)$ for each $i \in \{1, \ldots, n\}$.

Definition 2.1 (Alternating parity tree automata)**.** An *alternating parity tree automaton (APT)* is a quintuple $(Q, \Sigma, q_I, \delta, \Omega)$ where (i) Q is a finite set of states, (ii) Σ is a ranked alphabet; let $d = \mathbf{max}(\mathit{codom}(\Sigma))$, (iii) $q_I \in Q$ is called the initial state, (iv) $\delta : Q \times \Sigma \to \mathcal{P}(\mathcal{P}(\{1, \ldots, d\} \times Q))$ is called the transition function where $\delta(q, \mathsf{a}) \in \mathcal{P}(\mathcal{P}(\{1, \ldots, \Sigma(\mathsf{a})\} \times Q))$ holds for each $q \in Q$ and $\mathsf{a} \in \mathit{dom}(\Sigma)$ and (v) $\Omega : Q \to \{0, \ldots, M\}$ is called the priority function.

A *run tree* of an APT $\mathcal{A} = (Q, \Sigma, q_I, \delta, \Omega)$ over a Σ-labeled ranked tree T is a $(\mathit{dom}(T) \times Q)$-labeled tree R satisfying: (i) $\epsilon \in \mathit{dom}(R)$ and $R(\epsilon) = (\epsilon, q_I)$, (ii) for every $\beta \in \mathit{dom}(R)$ with $R(\beta) = (\alpha, q)$, there exists a set $S \in \delta(q, T(\alpha))$ such that, for each $(i, q') \in S$, there exists k such that $\beta k \in \mathit{dom}(R)$ and $R(\beta k) = (\alpha i, q')$. A ($\Sigma$-labeled ranked) tree T is *accepted* by $\mathcal{A}$ if there exists a run tree R of $\mathcal{A}$ over T such that every infinite path $i_1 i_2 \cdots$ of R satisfies the *parity condition* with respect to Ω, i.e., the largest number occurring infinitely often in $\Omega(R(\epsilon))\Omega(R(i_1))\Omega(R(i_1 i_2)) \ldots$ is even (here, we have extended Ω to a map from $\mathit{dom}(T) \times Q$ to $\{0, \ldots, M\}$ by $\Omega(\alpha, q) = \Omega(q)$). We write $\mathcal{L}(\mathcal{A})$ for the set of trees accepted by $\mathcal{A}$.

Intuitively, $\delta(q, \mathsf{a}) = \{S_1, \ldots, S_\ell\}$ means that upon reading a at state q, the automaton should choose some S_i and for each $(j, q') \in S_i$, spawn a thread to visit the j-th child with state q'. An input tree is accepted just if the largest priority visited infinitely often by each thread is even.

Example 2.2. Let $\Sigma_1 = \{\mathsf{a} \mapsto 2, \mathsf{b} \mapsto 1, \mathsf{c} \mapsto 1\}$ be a ranked alphabet. Consider the APT $\mathcal{A}_1 = (\{q_0, q_1, q_2\}, \Sigma_1, q_0, \delta_1, \Omega_1)$ where

$$\delta_1(q_0, \mathsf{a}) = \{\{(1, q_0)\}, \{(2, q_0)\}\}$$
$$\delta_1(q_1, \mathsf{a}) = \delta_1(q_2, \mathsf{a}) = \{\{(1, q_1), (2, q_1)\}\}$$
$$\delta_1(q_0, \mathsf{b}) = \delta_1(q_1, \mathsf{b}) = \delta_1(q_2, \mathsf{b}) = \{\{(1, q_1)\}\}$$
$$\delta_1(q_0, \mathsf{c}) = \{\{(1, q_0)\}\} \qquad \delta_1(q_1, \mathsf{c}) = \delta_1(q_2, \mathsf{c}) = \{\{(1, q_2)\}\}$$

and $\Omega_1 = \{q_0 \mapsto 1, q_1 \mapsto 1, q_2 \mapsto 2\}$. Upon reading a at the initial state q_0, the automaton chooses one of the children, and stays in state q_0 until it encounters b. Upon reading b, the automaton moves to state q_1. After that, it visits all the subtrees (notice the difference between $\delta_1(q_0, \mathsf{a})$ and $\delta_1(q_1, \mathsf{a})$), and each time it encounters c, it changes the state to q_2. Thus, a Σ_1-labeled tree T is accepted by $\mathcal{A}_1$ just if there exists a b-labeled node such that there is no ancestor node labeled with b, and every infinite path from the node contains infinitely many occurrences of c.

2.1.2 HORS$_\mathsf{Z}$

The set of *simple types*, ranged over by κ, is defined by:

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

The types $\star$ and $\mathtt{int}$ describe trees and integers respectively. For a ranked alphabet Σ, the set of *terms*, ranged over by t, is defined by:

$$t ::= x \mid n \mid t_1 \ \mathsf{op} \ t_2 \mid \mathbf{if}\ p(t_1', t_2')\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathsf{a}(t_1, \ldots, t_k) \mid t_1\ t_2 \mid \lambda x.t.$$

Here, n and x range over the sets of integers and variables respectively. The meta-variable a ranges over $\mathit{dom}(\Sigma)$. We consider only terms well-typed in a standard simple type system, and write $\mathbf{Terms}_{\Sigma,\mathcal{K},\kappa}$ for the set of terms that have type κ under type environment $\mathcal{K}$, i.e., $\{t \mid \mathcal{K} \vdash t : \kappa\}$, where the type judgment relation $\mathcal{K} \vdash t : \kappa$ is defined by the typing rules given in Figure 1.

A *HORS$_\mathsf{Z}$* is a quadruple $(\Sigma, \mathcal{K}, D, t_I)$ where Σ is a ranked alphabet, $\mathcal{K}$ is a type environment (on functions), $D = \{f_1 = t_1, \ldots, f_n = t_n\}$ is a set of function definitions such that $t_i \in \mathbf{Terms}_{\Sigma,\mathcal{K},\mathcal{K}(f_i)}$, and $t_I \in \mathbf{Terms}_{\Sigma,\mathcal{K},\star}$ is a term called the *start term*, which should not contain λ-abstractions. For the sake of simplicity, we assume that λ-abstractions occur only at the top-level in $t_1, \ldots, t_n$, i.e., t_i is of the form $\lambda x_1. \cdots \lambda x_k. t_i'$ where t_i' does not contain λ-abstractions, and furthermore, we assume that t_i' is of type $\star$. We use the meta-variable P for HORS$_\mathsf{Z}$. We often omit the first two components and just write (D, t_I) for HORS$_\mathsf{Z}$. We regard D as a map from function symbols to their bodies and write $D(f_i) = t_i$ and also write $\mathbf{funs}(D)$ for the set of function symbols $\{f_1, \ldots, f_n\}$. When $D(f) = \lambda\widetilde{x}.t$, we also write $f\ \widetilde{x} = t$ for the definition of f. We sometimes assume that the start term is a special function name $\mathbf{main} \in \mathbf{funs}(D)$,

$$\frac{}{\mathcal{K}, x : \eta \vdash x : \eta}$$

$$\frac{}{\mathcal{K} \vdash n : \mathtt{int}}$$

$$\frac{\mathcal{K} \vdash t_1 : \mathtt{int} \qquad \mathcal{K} \vdash t_2 : \mathtt{int}}{\mathcal{K} \vdash t_1 \ \mathsf{op} \ t_2 : \mathtt{int}}$$

$$\frac{k = \Sigma(\mathsf{a}) \qquad \forall i \in \{1, \ldots, k\}.\ \mathcal{K} \vdash t_i : \star}{\mathcal{K} \vdash \mathsf{a}(t_1, \ldots, t_k) : \star}$$

$$\frac{\forall i \in \{1,2\}.\ \mathcal{K} \vdash t_i' : \mathtt{int} \qquad \forall j \in \{1,2\}.\ \mathcal{K} \vdash t_j : \star}{\mathcal{K} \vdash \textbf{if}\ p(t_1', t_2')\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \star}$$

$$\frac{\mathcal{K} \vdash t_1 : \eta \to \kappa \qquad \mathcal{K} \vdash t_2 : \eta}{\mathcal{K} \vdash t_1 t_2 : \kappa}$$

$$\frac{\mathcal{K}, x : \eta \vdash t : \kappa}{\mathcal{K} \vdash \lambda x.t : \eta \to \kappa}$$

Figure 1. Typing rules for terms.

$$E(\text{evaluation contexts}) ::= [\,] \mid \mathsf{a}(t_1, \ldots, t_{i-1}, E, t_{i+1}, \ldots, t_k)$$

$$\frac{f = \lambda\widetilde{x}.u \in D \qquad |\widetilde{x}| = |\widetilde{t}|}{E[f\ \widetilde{t}] \to_D E[[\widetilde{t}/\widetilde{x}]u]}$$

$$\frac{(\llbracket t_1' \rrbracket, \llbracket t_2' \rrbracket) \in \llbracket p \rrbracket}{E[\textbf{if}\ p(t_1', t_2')\ \textbf{then}\ t_1\ \textbf{else}\ t_2] \to_D E[t_1]}$$

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

Figure 2. Reduction semantics.

where $\mathcal{K}(\textbf{main}) = \star$; Any HORS_Z (D, t_I) can be normalized to $(D \cup \{\textbf{main} = t_I\}, \textbf{main})$.

We define the reduction relation $\to_D$ on terms in Figure 2. In the rules, $\llbracket t \rrbracket$ represents the integer value of t, and $\llbracket p \rrbracket$ denotes the binary relation on integers represented by p. We write $\to_D^*$ for the reflexive and transitive closure of $\to_D$.

The tree generated by a HORS_Z (called the *value tree*) is defined in the same way as for HORS [30]. Given a term t, we define the (finite) tree $t^\perp$ by: $t^\perp = \mathsf{a}(t_1^\perp, \ldots, t_k^\perp)$ if $t = \mathsf{a}(t_1, \ldots, t_k)$, and $t^\perp = \perp$ otherwise. Let $\Sigma^\perp$ be $\Sigma \cup \{\perp \mapsto 0\}$, and define the partial order $\sqsubseteq$ on $\Sigma^\perp$-labeled trees as the least precongruence such that $\perp \sqsubseteq T$ for every $\Sigma^\perp$-labeled tree T. The value tree of HORS_Z $P = (\Sigma, \mathcal{K}, D, t_I)$, written $\textbf{Tree}(P)$, is then defined as $\bigsqcup\{t^\perp \mid t_I \to_D^* t\}$, where $\bigsqcup S$ denotes the least upper-bound (which is guaranteed to exist) of the elements of S with respect to $\sqsubseteq$.

The HORS_Z *model checking* is the problem of, given a HORS_Z P and an APT $\mathcal{A}$ as input, deciding whether $\textbf{Tree}(P)$ is accepted by $\mathcal{A}$. Note that unlike in the case of pure HORS, the HORS_Z model checking problem is undecidable.

For the sake of simplicity, we consider below only HORS_Z whose value trees contain no leaf (i.e. no node π such that $\Sigma(\textbf{Tree}(P)(\pi)) = 0$). This assumption does not lose generality, because the condition can be ensured by (i) replacing each body $\lambda\widetilde{x}.t$ of function definitions with $\lambda\widetilde{x}.\mathtt{call}(t)$ (where `call` is a unary symbol), (ii) replacing each nullary constant with a diverging term that generates an infinite tree, like `dummy` defined by $\mathtt{dummy} = d(\mathtt{dummy})$, and (iii) modifying APT accordingly.

Example 2.3. Consider the HORS_Z $P_1 = (\Sigma_1, \mathcal{K}_1, D_1, f\ 0)$ where Σ_1 is as in Example 2.2, $\mathcal{K}_1 = (f : \mathtt{int} \to \star, g : \mathtt{int} \to \star)$, and:

$$D_1 = \{f\ x = \mathsf{a}(f\ (x+1), g\ x),\\ g\ x = \textbf{if}\ x = 0\ \textbf{then}\ \mathsf{c}(g\ 0)\ \textbf{else}\ \mathsf{b}(g\ (x-1))\}.$$

The APT $\mathcal{A}_1$ in Example 2.2 accepts the value tree of P_1. In fact, the value tree is of the form $\mathsf{a}(\mathsf{a}(\cdots, \mathsf{bc}^\omega), \mathsf{c}^\omega)$; so, `b` occurs in the tree, and `c` occurs infinitely often below the rightmost occurrence of `b`.

□

Example 2.4. Consider the following OCaml-like program.

```
let rec repeat f x =
  if x<=0 then ()
  else if * then repeat f (f x) else repeat f (x-1)
in let y = input() in
  repeat (fun x->x-y) n
```

Here, $*$ denotes a non-deterministic Boolean value, and n is an integer constant. The program takes an integer y as a user's input, and calls `repeat` $(\lambda x.x - y)\ n$. Suppose we wish to verify that if a user provides an appropriate input, then the program eventually terminates in any non-deterministic choice of Booleans (which is indeed the case if a user provides a positive integer). Then, we can represent the program as the HORS_Z $P_2 = (\Sigma_2, \mathcal{K}_2, D_2, \mathtt{input}\ 0\ (g\ n))$ where $\Sigma_2 = \{\mathsf{a} \mapsto 2, \mathsf{c} \mapsto 1, \mathsf{d} \mapsto 2\}$, $\mathcal{K}_2$ is

$$\mathtt{repeat} : (\mathtt{int} \to (\mathtt{int} \to \star) \to \star) \to \mathtt{int} \to \star,\\ \mathtt{fin} : \star,\ \mathtt{sub} : \mathtt{int} \to \mathtt{int} \to (\mathtt{int} \to \star) \to \star,\\ g : \mathtt{int} \to \mathtt{int} \to \star,\ \mathtt{input} : \mathtt{int} \to (\mathtt{int} \to \star) \to \star,$$

and D_2 consists of:

$$\mathtt{repeat}\ f\ x = \textbf{if}\ x \le 0\ \textbf{then}\ \mathtt{fin}\ \textbf{else}\\ \qquad \mathsf{d}(f\ x\ (\mathtt{repeat}\ f), \mathtt{repeat}\ f\ (x-1))\\ \mathtt{fin} = \mathsf{c}(\mathtt{fin})\\ \mathtt{sub}\ x\ y\ k = k\ (y - x)\\ g\ z\ x = \mathtt{repeat}\ (\mathtt{sub}\ x)\ z\\ \mathtt{input}\ x\ k = \mathsf{a}(k\ x, \mathtt{input}\ (x+1)\ k).$$

Here, a represents an angelic non-deterministic choice controlled by a user,[2] d represents a demonic non-deterministic choice made by the program, and c denotes termination. Thus, the goal of model checking is to check that if an appropriate branch is chosen at each a-node, c occurs eventually no matter which branch is chosen at each d-node. That property can be expressed by using the APT $\mathcal{A}_2 = (\{q_0\}, \Sigma_2, q_0, \delta_2, \Omega_2)$ where:

$$\delta_2(q_0, \mathsf{a}) = \{\{(1, q_0)\}, \{(2, q_0)\}\}$$
$$\delta_2(q_0, \mathsf{d}) = \{\{(1, q_0), (2, q_0)\}\}$$
$$\delta_2(q_0, \mathsf{c}) = \{\emptyset\}$$
$$\Omega_2(q_0) = 1$$

□

2.2 HFL$_Z$ Validity Checking

We now define the target problem: validity checking of modal-free HFL$_Z$ formulas. The *modal-free HFL$_Z$* is a fragment of HFL$_Z$ [22] without modal operators, where HFL$_Z$ is an extension of higher-order modal fixpoint logic (HFL) [33] with integer predicates. Henceforth, we often drop the adjective "modal-free" and just call the logic HFL$_Z$.

The syntax of *modal-free HFL$_Z$ formulas* is given by:

$$\varphi \text{ (formulas)} ::= n \mid \varphi_1 \ \mathsf{op} \ \varphi_2 \mid \mathtt{true} \mid \mathtt{false} \mid p(\varphi_1, \varphi_2) \mid X \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mu X^\tau.\varphi \mid \nu X^\tau.\varphi \mid \lambda X^\sigma.\varphi \mid \varphi_1 \ \varphi_2$$
$$\tau \text{ (types)} ::= \bullet \mid \sigma \to \tau \qquad \sigma \text{ (extended types)} ::= \tau \mid \mathtt{int}$$

Here, X ranges over a set of variables. The type $\bullet$ describes propositions, and $\sigma \to \tau$ describes functions from values of type σ to those of τ; actually, according to the syntax, any type must be of the form $\sigma_1 \to \cdots \to \sigma_k \to \bullet$, so it represents the type of a (possibly higher-order) predicate. The formulas $\mu X^\tau.\varphi$ and $\nu X^\tau.\varphi$ respectively represent the least and greatest fixpoints of the function $\lambda X^\tau.\varphi$ (of type $\tau \to \tau$). We often omit type annotations and just write $\mu X.\varphi$, $\nu X.\varphi$ and $\lambda X.\varphi$ for $\mu X^\tau.\varphi$, $\nu X^\tau.\varphi$ and $\lambda X^\sigma.\varphi$ respectively. We do not have the negation operator $\neg$, but for any closed formula φ of type $\bullet$, its negation can be expressed without $\neg$ [27].

As usual, we restrict the syntax of formulas using a simple type system. The type judgment relation $\Delta \vdash_{\mathsf{H}} \varphi : \tau$ is obtained from that of the simply-typed λ-calculus, by regarding φ as a simply-typed λ-term constructed from constants: $\mathtt{true}, \mathtt{false} : \bullet$; $\wedge, \vee : \bullet \to \bullet \to \bullet$; $\mu^\tau, \nu^\tau : (\tau \to \tau) \to \tau$; etc. (here, $\mu X^\tau.\varphi$ and $\nu X^\tau.\varphi$ are considered abbreviated forms of $\mu^\tau(\lambda X^\tau.\varphi)$ and $\nu^\tau(\lambda X^\tau.\varphi)$ respectively). The typing rules are given in Figure 3. Henceforth, we consider only well-typed formulas.

[2] For the sake of simplicity, we have restricted the user's input to non-negative integers.

$$\Delta \vdash_{\mathsf{H}} n : \mathtt{int} \ \text{(HT-Int)} \qquad \frac{\Delta \vdash_{\mathsf{H}} \varphi_1 : \bullet \qquad \Delta \vdash_{\mathsf{H}} \varphi_2 : \bullet}{\Delta \vdash_{\mathsf{H}} \varphi_1 \wedge \varphi_2 : \bullet} \ \text{(HT-And)}$$

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

$$\Delta \vdash_{\mathsf{H}} \mathtt{false} : \bullet \ \text{(HT-False)} \qquad \frac{\Delta, X : \tau \vdash_{\mathsf{H}} \varphi : \tau}{\Delta \vdash_{\mathsf{H}} \mu X^\tau.\ \varphi : \tau} \ \text{(HT-Mu)}$$

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

$$\frac{\Delta \vdash_{\mathsf{H}} \varphi_1 : \mathtt{int} \qquad \Delta \vdash_{\mathsf{H}} \varphi_2 : \mathtt{int}}{\Delta \vdash_{\mathsf{H}} \varphi_1 \ \mathsf{op} \ \varphi_2 : \mathtt{int}} \ \text{(HT-Op)} \qquad \frac{\Delta, X : \tau \vdash_{\mathsf{H}} \varphi : \tau}{\Delta \vdash_{\mathsf{H}} \nu X^\tau.\ \varphi : \tau} \ \text{(HT-Nu)}$$

$$\frac{\Delta \vdash_{\mathsf{H}} \varphi_1 : \mathtt{int} \qquad \Delta \vdash_{\mathsf{H}} \varphi_2 : \mathtt{int}}{\Delta \vdash_{\mathsf{H}} p(\varphi_1, \varphi_2) : \bullet} \ \text{(HT-Pred)} \qquad \frac{\Delta, X : \sigma \vdash_{\mathsf{H}} \varphi : \tau}{\Delta \vdash_{\mathsf{H}} \lambda X^\sigma.\ \varphi : \sigma \to \tau} \ \text{(HT-Abs)}$$

$$\frac{\Delta \vdash_{\mathsf{H}} \varphi_1 : \bullet \qquad \Delta \vdash_{\mathsf{H}} \varphi_2 : \bullet}{\Delta \vdash_{\mathsf{H}} \varphi_1 \vee \varphi_2 : \bullet} \ \text{(HT-Or)} \qquad \frac{\Delta \vdash_{\mathsf{H}} \varphi_1 : \sigma \to \tau \qquad \Delta \vdash_{\mathsf{H}} \varphi_2 : \sigma}{\Delta \vdash_{\mathsf{H}} \varphi_1 \ \varphi_2 : \tau} \ \text{(HT-App)}$$

Figure 3. Typing rules for modal-free HFL$_Z$ formulas.

Now we define the semantics of modal-free HFL$_Z$ formulas. For each (extended) type σ, we define the partially ordered set $(\mathcal{D}_\sigma, \sqsubseteq_\sigma)$ by induction on types.

$$\mathcal{D}_\bullet = \{\mathtt{false}, \mathtt{true}\} \qquad \sqsubseteq_\bullet = \{(\mathtt{false}, \mathtt{true})\}^*$$
$$\mathcal{D}_{\mathtt{int}} = \mathbf{Z} \qquad \sqsubseteq_{\mathtt{int}} = \{(n, n) \mid n \in \mathbf{Z}\}$$
$$\mathcal{D}_{\sigma\to\tau} = \{f \in \mathcal{D}_\sigma \to \mathcal{D}_\tau \mid \forall x, y \in \mathcal{D}_\sigma.(x \sqsubseteq_\sigma y \Rightarrow f\ x \sqsubseteq_\tau f\ y)\}$$
$$\sqsubseteq_{\sigma\to\tau} = \{(f, g) \mid \forall x \in \mathcal{D}_\sigma.f(x) \sqsubseteq_\tau g(x)\}.$$

Note that for each type $\tau(\neq \mathtt{int})$, $(\mathcal{D}_\tau, \sqsubseteq_\tau)$ forms a complete lattice. We write $\bot_\tau$ and $\top_\tau$ for the least and greatest elements of $\mathcal{D}_\tau$ respectively.

For a type environment Δ, we write $[\![\Delta]\!]$ for the set of functions that maps X to an element of $\mathcal{D}_\sigma$ for each $X{:}\sigma \in \Delta$. For an HFL$_Z$ formula φ such that $\Delta \vdash_{\mathsf{H}} \varphi : \sigma$, we define the interpretation $[\![\Delta \vdash_{\mathsf{H}} \varphi : \sigma]\!]$ as a map from $[\![\Delta]\!]$ to D_σ by induction on the (unique) derivation of $\Delta \vdash_{\mathsf{H}} \varphi{:}\sigma$, as given in Figure 4. Here, $[\![\mathsf{op}]\!]$ denotes the binary function on integers represented by op. The operators $\mathbf{lfp}_\tau$ and $\mathbf{gfp}_\tau$ are the least and greatest fixpoint operators defined by:

$$\mathbf{lfp}_\tau(f) = \textstyle\bigsqcap_\tau \{x \in \mathcal{D}_\tau \mid f(x) \sqsubseteq_\tau x\}$$
$$\mathbf{gfp}_\tau(f) = \textstyle\bigsqcup_\tau \{x \in \mathcal{D}_\tau \mid x \sqsubseteq_\tau f(x)\}$$

$$[\![\Delta \vdash_{\mathsf{H}} n : \mathtt{int}]\!](\rho) = n$$
$$[\![\Delta \vdash_{\mathsf{H}} \mathtt{true} : \bullet]\!](\rho) = \mathtt{true} \qquad [\![\Delta \vdash_{\mathsf{H}} \mathtt{false} : \bullet]\!](\rho) = \mathtt{false}$$
$$[\![\Delta \vdash_{\mathsf{H}} \varphi_1 \ \mathsf{op} \ \varphi_2 : \mathtt{int}]\!](\rho) =$$
$$([\![\Delta \vdash_{\mathsf{H}} \varphi_1 : \mathtt{int}]\!](\rho))[\![\mathsf{op}]\!]([\![\Delta \vdash_{\mathsf{H}} \varphi_2 : \mathtt{int}]\!](\rho))$$
$$[\![\Delta \vdash_{\mathsf{H}} p(\varphi_1, \varphi_2) : \bullet]\!](\rho) =$$
$$(([\![\Delta \vdash_{\mathsf{H}} \varphi_1 : \mathtt{int}]\!](\rho), [\![\Delta \vdash_{\mathsf{H}} \varphi_2 : \mathtt{int}]\!](\rho)) \in [\![p]\!])$$
$$[\![\Delta, X : \sigma \vdash_{\mathsf{H}} X : \sigma]\!](\rho) = \rho(X)$$
$$[\![\Delta \vdash_{\mathsf{H}} \varphi_1 \vee \varphi_2 : \bullet]\!](\rho) = [\![\Delta \vdash_{\mathsf{H}} \varphi_1 : \bullet]\!](\rho) \sqcup [\![\Delta \vdash_{\mathsf{H}} \varphi_2 : \bullet]\!](\rho)$$
$$[\![\Delta \vdash_{\mathsf{H}} \varphi_1 \wedge \varphi_2 : \bullet]\!](\rho) = [\![\Delta \vdash_{\mathsf{H}} \varphi_1 : \bullet]\!](\rho) \sqcap [\![\Delta \vdash_{\mathsf{H}} \varphi_2 : \bullet]\!](\rho)$$
$$[\![\Delta \vdash_{\mathsf{H}} \mu X^{\tau}.\varphi : \tau]\!](\rho) = \mathbf{lfp}_{\tau}([\![\Delta \vdash_{\mathsf{H}} \lambda X^{\tau}.\ \varphi : \tau \to \tau]\!](\rho))$$
$$[\![\Delta \vdash_{\mathsf{H}} \nu X^{\tau}.\varphi : \tau]\!](\rho) = \mathbf{gfp}_{\tau}([\![\Delta \vdash_{\mathsf{H}} \lambda X^{\tau}.\ \varphi : \tau \to \tau]\!](\rho))$$
$$[\![\Delta \vdash_{\mathsf{H}} \lambda X^{\sigma}.\ \varphi : \sigma \to \tau]\!](\rho)$$
$$= \{(v, [\![\Delta, X : \sigma \vdash_{\mathsf{H}} \varphi : \tau]\!](\rho[X \mapsto v])) \mid v \in \mathcal{D}_{\sigma}\}$$
$$[\![\Delta \vdash_{\mathsf{H}} \varphi_1\ \varphi_2 : \tau]\!](\rho) = [\![\Delta \vdash_{\mathsf{H}} \varphi_1 : \sigma \to \tau]\!](\rho)([\![\Delta \vdash_{\mathsf{H}} \varphi_2 : \sigma]\!](\rho))$$

Figure 4. Semantics of $\mathrm{HFL}_{\mathrm{Z}}$ formulas

Here $\bigsqcap_{\tau}$ and $\bigsqcup_{\tau}$ denotes the greatest lower bound and the least upper bound respectively. If φ is a closed formula, we just write $[\![\varphi]\!]$ for $[\![\emptyset \vdash_{\mathsf{H}} \varphi : \sigma]\!](\emptyset)$. We write $\models \varphi$ if $[\![\varphi]\!] = \mathtt{true}$.

Example 2.5. The formulas $\mu X^{\bullet}.X$ and $\nu X^{\bullet}.X$ are equivalent to `false` and `true` respectively. Let φ be the formula $\mu X^{\mathtt{int}\to\bullet}.\lambda y.\, y = 0 \vee X(y-2)$. Then $\models \varphi(n)$ if and only if n is a non-negative even number. Let ψ be a formula of type $\mathtt{int} \to \bullet$. Then $(\nu X^{\mathtt{int}\to\bullet}.\lambda y.\psi(y) \wedge X(y+1) \wedge X(y-1))0$ means that $\psi(n)$ holds for every integer n; thus, we can express quantifiers.

The $\mathrm{HFL}_{\mathrm{Z}}$ *validity checking problem* is the problem of deciding, given a closed $\mathrm{HFL}_{\mathrm{Z}}$ formula φ of type $\bullet$, whether $\models \varphi$. The $\mathrm{HFL}_{\mathrm{Z}}$ validity checking problem is undecidable in general. If φ is a *pure* closed HFL formula, i.e., if it does not contain any subterms of type `int`, however, the validity checking problem is decidable (which is obvious, since the semantic domain D_{σ} is finite if σ does not contain `int`).[3]

For technical convenience, we often express an HFL formula as a sequence of fixpoint equations, called a *hierarchical equation system* (HES) [19].

Definition 2.6 (Hierarchical equation system (HES))**.** A *hierarchical equation system* (HES) is a pair $(\mathcal{E}, \varphi)$, where $\mathcal{E}$ is a sequence of fixpoint equations of the form $X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n$. Here, $\alpha_i \in \{\mu, \nu\}$ for each $i \in \{1, \ldots, n\}$, and we require that (i) $X_1 : \tau_1, \ldots, X_n : \tau_n \vdash_{\mathsf{H}} \varphi_i : \tau_i$ holds for each $i \in \{1, \ldots, n\}$, and (ii) none of $\varphi_i (i \in \{1, \ldots, n\})$ and φ contains any fixpoint operators. We write $\mathcal{E}(X_i^{\tau_i})$ for φ_i.

The HFL formula denoted by an HES $\Phi = (\mathcal{E}, \varphi)$ is $\mathit{toHFL}(\mathcal{E}, \varphi)$ defined inductively by:

$$\mathit{toHFL}(\epsilon, \varphi) = \varphi$$
$$\mathit{toHFL}(\mathcal{E}; X^{\tau} =_{\alpha} \varphi', \varphi) =$$
$$\quad \mathit{toHFL}([\alpha X^{\tau}.\varphi'/X]\mathcal{E}, [\alpha X^{\tau}.\varphi'/X]\varphi).$$

We write $[\![\Phi]\!]$ for $\mathit{toHFL}([\![\Phi]\!])$ and $\models \Phi$ for $\models \mathit{toHFL}(\Phi)$. Every $\mathrm{HFL}_{\mathrm{Z}}$ formula can be transformed to an equivalent HES. For example, $(\nu X.\lambda x.(\mu Y.\lambda y.(y = 0 \vee Y(y-2)))(x) \wedge X(x+2))(0)$ can be expressed as

$$((X^{\mathtt{int}\to\bullet} =_{\nu} \lambda x.Y(x) \wedge X(x+2);$$
$$\quad Y^{\mathtt{int}\to\bullet} =_{\mu} \lambda y.y = 0 \vee Y(y-2)), \qquad X\,0).$$

Below, we often write $X^{\tau} x_1 \ldots x_k =_{\alpha} \varphi$ for $X^{\tau} =_{\alpha} \lambda x_1. \ldots .\lambda x_k.\varphi$. Also, we often omit type annotations and just write $X =_{\alpha} \varphi$ for $X^{\tau} =_{\alpha} \varphi$.

3 Reduction

This section presents our reduction from $\mathrm{HORS}_{\mathrm{Z}}$ model checking to $\mathrm{HFL}_{\mathrm{Z}}$ validity checking.

Given a $\mathrm{HORS}_{\mathrm{Z}}$ P and an automaton $\mathcal{A}$, we transform each $\mathrm{HORS}_{\mathrm{Z}}$ term t of type $\star$ to an $\mathrm{HFL}_{\mathrm{Z}}$ formula $[t]_q$ that intuitively says "the tree generated by t is accepted by $\mathcal{A}$ from state q." For example, recall the automaton $\mathcal{A}_1$ in Example 2.2, and consider the term $t = \mathsf{a}(t_1, t_2)$. Since $\delta_1(q_0, \mathsf{a}) = \{\{(1, q_0)\}, \{(2, q_0)\}\}$, the formula $[t]_{q_0}$ that means "the tree generated by t is accepted by $\mathcal{A}$ from q_0" is expressed by $[t_1]_{q_0} \vee [t_2]_{q_0}$. In contrast, $[t]_{q_1}$ is expressed by $[t_1]_{q_1} \wedge [t_2]_{q_1}$, since $\delta_1(q_1, \mathsf{a}) = \{\{(1, q_1), (2, q_1)\}\}$, which means that in order for a tree to be accepted from state q_1, both of its children should be accepted from state q_1.

The explanation above suggests that the translation from a $\mathrm{HORS}_{\mathrm{Z}}$ term to a formula should be parameterized by a state of the automaton in which the root node of the tree generated by the term is visited. Thus, each function symbol and formal argument should also be parameterized by a state. In addition, we also need to parameterize the translation with a priority, to respect the parity acceptance condition. To see such a need, consider the same automaton $\mathcal{A}_1$, and the following two functions for generating infinite trees b^{ω} and $(\mathsf{cb})^{\omega}$ respectively.

$$f = \mathsf{b}(f) \qquad g = \mathsf{c}(\mathsf{b}(g)).$$

Note that the tree generated by g should be accepted from q_1, whereas the one by f should not. If the translation were parameterized only with a state, we would get:

$$f_{q_1} = [\mathsf{b}(f)]_{q_1} = f_{q_1}$$
$$g_{q_1} = [\mathsf{c}(\mathsf{b}(g))x]_{q_1} = [\mathsf{b}(g)]_{q_2} = g_{q_1}.$$

Thus, we would get equations (of HES) of the same form: $f_{q_1} =_{\alpha} f_{q_1}$ and $g_{q_1} =_{\alpha'} g_{q_1}$; hence there would be no way to distinguish between them. To address this problem, we

[3] Recall that we are considering the modal-free fragment; in the presence of modal operators, the validity checking problem (of deciding whether a given formula is satisfied by all structures) is undecidable even for pure HFL [33].

parameterize the translation (and also each function symbol and variable) with the largest priority of the states visited since the last recursive call, as in [19]. For the example above, the actual translation for the terms on the righthand side becomes

$$[\mathsf{b}(f)]_{q_1,0} = f_{q_1,1}$$
$$[\mathsf{c}(\mathsf{b}(g))]_{q_1,0} = [\mathsf{b}(g)]_{q_2,2} = g_{q_1,2}.$$

Here, the body $\mathsf{c}(\mathsf{b}(g))$ of g is translated to $g_{q_1,2}$, because the visited states are q_1 and q_2, whose largest priority is 2. We then bind function symbols parameterized with even priorities by ν and those with odd priorities by μ. Thus, f is translated to $\mu f_{q_1,1}.f_{q_1,1}$, which is equivalent to `false`, whereas g is translated to $\nu g_{q_1,2}.g_{q_1,2}$, which is equivalent to `true`.

Based on the ideas above, we now formally define the translation from programs to HFL_Z formulas. We fix an APT $\mathcal{A} = (\Sigma, Q, \delta_\mathcal{A}, q_1, \Omega)$, where $Q = \{q_1, \ldots, q_k\}$ and $\max(codom(\Omega)) = M$.

For a HORS_Z term t, a state q and a priority $m \in \{0, \ldots, M\}$, we define the HFL_Z formula $[t]_{q,m}$ by:

$$\begin{aligned}
&[n]_{q,m} = n\\
&[x]_{q,m} = x_{q,m}\\
&[t_1 \ \mathsf{op}\ t_2]_{q,m} = [t_1]_{q,m} \ \mathsf{op}\ [t_2]_{q,m}\\
&[t_1\, t_2]_{q,m} =\\
&\quad [t_1]_{q,m}\, [t_2]_{q_1,\max(m,0)} \cdots [t_2]_{q_1,\max(m,M)} \cdots\\
&\qquad [t_2]_{q_k,\max(m,0)} \cdots [t_2]_{q_k,\max(m,M)}\\
&[\lambda x.t]_{q,m} = \lambda x_{q_1,0}. \cdots \lambda x_{q_1,M}. \cdots \lambda x_{q_k,0}. \cdots \lambda x_{q_k,M}.[t]_{q,m}\\
&[\mathbf{if}\ p(t_1', t_2')\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]_{q,m} =\\
&\quad (p([t_1']_{q,m}, [t_2']_{q,m}) \wedge [t_1]_{q,m})\\
&\quad \vee (\neg p([t_1']_{q,m}, [t_2']_{q,m}) \wedge [t_2]_{q,m})\\
&[\mathsf{a}(t_1, \ldots, t_\ell)]_{q,m} =\\
&\quad \bigvee_{S_i \in \delta_\mathcal{A}(q,a)} (\bigwedge_{(d_{i,j}, q_{i,j}) \in S_i} [t_{d_{i,j}}]_{q_{i,j},\max(m,\Omega(q_{i,j}))}).
\end{aligned}$$

Here, x in the second clause ranges over both the set of function symbols and variables (bound by λ). As explained above, each variable (or function symbol) is parameterized with a state and a priority; thus, in applications and λ-abstractions, arguments are replicated for each pair $(q_i, m_j) \in Q \times \{0, \ldots, M\}$.[4] In the translation of applications, the priority used to translate t_2 is raised up to m. Note that the i-th (counted from 0) copy of t_2 is used by the function $[t_1]_{q,m}$ when the largest priority visited since the function is called is i; since the priority m has been visited already in the calling context, t_2 is parameterized by $\max(m, i)$. In the last clause for tree constructors, the nesting of disjunctions and conjunctions reflects the meaning of the transition function (recall that $\delta(q, a) = \{S_1, \ldots, S_k\}$ means that *one of* S_i's should be picked, and for *every* $(d_{i,j}, q_{i,j}) \in S_i$, the $d_{i,j}$-th child should be visited with state $q_{i,j}$); the parameters q, m are updated accordingly.

[4] Actually, we need not to replicate arguments of type `int`; here we replicate them for the sake of simplicity of the definition of the translation (to avoid a case analysis on the types of arguments). In some of the examples given later, we avoid replications of integers.

Let $P = (\Sigma, \mathcal{K}, D, t_I)$ be a HORS_Z, where $D = \{f_1 = t_1, \ldots, f_n = t_n\}$. Let $\mathcal{E}_{i,j}$ be the sequence of equations:

$$f_{i,q_1,j} =_{\alpha(j)} [t_i]_{q_1,0}; \cdots; f_{i,q_k,j} =_{\alpha(j)} [t_i]_{q_k,0}$$

where $\alpha(j) = \nu$ if j is even and $\alpha(j) = \mu$ otherwise. We define the HES $\Phi_{P,\mathcal{A}}$ as $(\mathcal{E}, [t_I]_{q_1,0})$ where

$$\mathcal{E} = (\mathcal{E}_{1,M}; \cdots\ \mathcal{E}_{n,M}; \cdots\ \mathcal{E}_{1,0}; \cdots\ \mathcal{E}_{n,0}).$$

Note that the righthand side of the equation for each function $f_{i,q_\ell,j}$ is translated using priority 0, irrespectively of j. This is because the priority represents the largest one since the last recursive call; since $f_{i,q_\ell,j}$ has just been called when the righthand side is evaluated, the largest priority is 0.

Example 3.1. Recall APT $\mathcal{A}_1$ in Example 2.2 and HORS_Z P_1 in Example 2.3. $\Phi_{P_1,\mathcal{A}_1}$ is $(\mathcal{E}, f_{q_0,0}\, 0)$, where $\mathcal{E}$ is:

$$\begin{aligned}
&f_{q_0,2} =_\nu t_{f,q_0}; f_{q_1,2} =_\nu t_{f,q_1}; f_{q_2,2} =_\nu t_{f,q_2};\\
&g_{q_0,2} =_\nu t_{g,q_0}; g_{q_1,2} =_\nu t_{g,q_1}; g_{q_2,2} =_\nu t_{g,q_2};\\
&f_{q_0,1} =_\mu t_{f,q_0}; f_{q_1,1} =_\mu t_{f,q_1}; f_{q_2,1} =_\mu t_{f,q_2};\\
&g_{q_0,1} =_\mu t_{g,q_0}; g_{q_1,1} =_\mu t_{g,q_1}; g_{q_2,1} =_\mu t_{g,q_2};\\
&f_{q_0,0} =_\nu t_{f,q_0}; f_{q_1,0} =_\nu t_{f,q_1}; f_{q_2,0} =_\nu t_{f,q_2};\\
&g_{q_0,0} =_\nu t_{g,q_0}; g_{q_1,0} =_\nu t_{g,q_1}; g_{q_2,0} =_\nu t_{g,q_2}
\end{aligned}$$

where

$$\begin{aligned}
t_{f,q_0} &= \lambda\widetilde{x}.(f_{q_0,1}\, \widetilde{x+1}) \vee (g_{q_0,1}\, \widetilde{x}')\\
t_{f,q_1} &= t_{f,q_2} = \lambda\widetilde{x}.(f_{q_1,1}\, \widetilde{x+1}) \wedge (g_{q_1,1}\, \widetilde{x}')\\
t_{g,q_0} &= \lambda\widetilde{x}.(x_{q_0,0} = 0 \wedge g_{q_0,1}\, \widetilde{0}) \vee (x_{q_0,0} \neq 0 \wedge g_{q_1,1} \widetilde{x-1})\\
t_{g,q_1} &= \lambda\widetilde{x}.(x_{q_1,0} = 0 \wedge g_{q_2,2}\, \widetilde{0}) \vee (x_{q_1,0} \neq 0 \wedge g_{q_1,1}\, \widetilde{x-1})\\
t_{g,q_2} &= \lambda\widetilde{x}.(x_{q_2,0} = 0 \wedge g_{q_2,2}\, \widetilde{0}) \vee (x_{q_2,0} \neq 0 \wedge g_{q_1,1}\, \widetilde{x-1}).
\end{aligned}$$

Here, $\widetilde{0}$, $\widetilde{x}$ and $\widetilde{x}'$ denote the sequences:
$\underbrace{0 \cdots 0}_{9}$, $x_{q_0,0}\, x_{q_0,1}\, x_{q_0,2}\, x_{q_1,0}\, x_{q_1,1}\, x_{q_1,2}\, x_{q_2,0}\, x_{q_2,1}\, x_{q_2,2}$,
and $x_{q_0,1}\, x_{q_0,1}\, x_{q_0,2}\, x_{q_1,1}\, x_{q_1,1}\, x_{q_1,2}\, x_{q_2,1}\, x_{q_2,1}\, x_{q_2,2}$ respectively; $\widetilde{x+1}$ denotes the sequence consisting of $x_{q_i,\max(1,j)} + 1$; similarly for $\widetilde{x-1}$. By eliminating unused functions and avoiding unnecessary replications of integer arguments, we obtain a simplified version: $(\mathcal{E}', f_{q_0,1}\, 0)$ where $\mathcal{E}'$ is:

$$\begin{aligned}
&g_{q_2,2} =_\nu \lambda x.(x = 0 \wedge g_{q_2,2}\, 0) \vee (x \neq 0 \wedge g_{q_1,1}\, (x-1));\\
&f_{q_0,1} =_\mu \lambda x.(f_{q_0,1}\, (x+1)) \vee (g_{q_0,1}\, x)\\
&g_{q_0,1} =_\mu \lambda x.(x = 0 \wedge g_{q_0,1}\, 0) \vee (x \neq 0 \wedge g_{q_1,1}(x-1));\\
&g_{q_1,1} =_\mu \lambda x.(x = 0 \wedge g_{q_2,2}\, 0) \vee (x \neq 0 \wedge g_{q_1,1}\, (x-1)).
\end{aligned}$$

Since $g_{q_2,2}\, 0$ is equivalent to `true`, so is $g_{q_1,1}\, 0$, hence also is $g_{q_0,1}\, 1$. Thus, $f_{q_0,1}\, 0 \equiv (f_{q_0,1}\, 1 \vee g_{q_0,1}\, 0) \equiv (f_{q_0,1}\, 2 \vee g_{q_0,1}\, 1 \vee g_{q_0,1}\, 0)$ is also equivalent to `true`. □

Example 3.2. Consider $\mathcal{A}_1$ in Example 2.2, and the HORS_Z $P_3 = (\Sigma_1, \mathcal{K}_3, D_3, f\, 0)$, where: Σ_1 is as in Example 2.2, and: $\mathcal{K}_3 = (f : \mathsf{int} \to \star, g : \mathsf{int} \to (\star \to \star) \to \star, h : \star \to \star)$, and D_3 consists of the following function definitions:

$$\begin{aligned}
&f\ x = \mathsf{a}(f(x+1), g\ x\ h),\\
&g\ x\ y = \mathbf{if}\ x = 0\ \mathbf{then}\ y(g\ 0\ y)\ \mathbf{else}\ \mathsf{b}(g\,(x-1)\,y),\\
&h\ z = \mathsf{c}(z).
\end{aligned}$$

This is a slightly modified version of P_1, which generates the same tree.[5] $\Phi_{P_2,\mathcal{A}_1}$ is $(\mathcal{E}_2, f_{q_0,0}\,0)$, where $\mathcal{E}_2$ is:

$$
\begin{array}{l}
f_{q_0,2} =_\nu \lambda x.(f_{q_0,1}\,(x+1) \vee g_{q_0,1}\,x\,\widetilde{h_1});\\
f_{q_1,2} =_\nu \lambda x.(f_{q_1,1}\,(x+1) \wedge g_{q_1,1}\,x\,\widetilde{h_1}); f_{q_2,2} =_\nu \cdots;\\
g_{q_0,2} =_\nu\\
\quad \lambda x.\lambda\widetilde{y}.(x = 0 \wedge y_{q_0,0}\,\widetilde{g\,0\,y}) \vee (x \neq 0 \wedge g_{q_1,1}(x-1)\widetilde{y_1});\\
g_{q_1,2} =_\nu\\
\quad \lambda x.\lambda\widetilde{y}.(x = 0 \wedge y_{q_1,0}\,\widetilde{g\,0\,y}) \vee (x \neq 0 \wedge g_{q_1,1}(x-1)\widetilde{y_1});\\
g_{q_2,2} =_\nu\\
\quad \lambda x.\lambda\widetilde{y}.(x = 0 \wedge y_{q_2,0}\,\widetilde{g\,0\,y}) \vee (x \neq 0 \wedge g_{q_1,1}(x-1)\widetilde{y_1});\\
h_{q_0,2}\,\widetilde{z} =_\nu z_{q_0,1}; h_{q_1,2}\,\widetilde{z} =_\nu z_{q_2,2}; h_{q_2,2}\,\widetilde{z} =_\nu z_{q_2,2}\\
f_{q_0,1} =_\mu \lambda x.(f_{q_0,1}\,(x+1) \vee g_{q_0,1}\,x\,\widetilde{h_1});\\
f_{q_1,1} =_\mu \lambda x.(f_{q_1,1}\,(x+1) \wedge g_{q_1,1}\,x\,\widetilde{h_1});\\
\cdots \text{ (the bodies of } f_{q,i}, g_{q,i}, h_{q,i} \text{ are the same as}\\
\qquad \text{those of } f_{q,2}, g_{q,2}, h_{q,2})
\end{array}
$$

Here, $\widetilde{y}$ denotes the sequence:

$$y_{q_0,0}\,y_{q_0,1}\,y_{q_0,2}\,y_{q_1,0}\,y_{q_1,1}\,y_{q_1,2}\,y_{q_2,0}\,y_{q_2,1}\,y_{q_2,2}$$

and similarly for $\widetilde{z}$; $\widetilde{h_1}$ denotes

$$h_{q_0,1}\,h_{q_0,1}\,h_{q_0,2}\,h_{q_1,1}\,h_{q_1,1}\,h_{q_1,2}\,h_{q_2,1}\,h_{q_2,1}\,h_{q_2,2}$$

(note that the priority is at least 1; similarly for $\widetilde{y_1}$); $\widetilde{g\,0\,y}$ denotes the sequence consisting of the terms: $g_{q_i,m}\,0\,\widetilde{y_m}$ for $i, m \in \{0,1,2\}$, where $\widetilde{y_m}$ denotes the sequence:

$$
\begin{array}{l}
y_{q_0,\max(m,0)}\,y_{q_0,\max(m,1)}\,y_{q_0,\max(m,2)}\,\cdots\\
y_{q_2,\max(m,0)}\,y_{q_2,\max(m,1)}\,y_{q_2,\max(m,2)}.
\end{array}
$$

Note that we have avoided unnecessary replications of integer arguments above. □

As indicated in the examples above, the result of the translation may contain many redundant arguments. Some of them can be removed by using a standard technique for useless variable elimination [2, 16, 34]. The output of the translation is significantly simpler if an APT consists of a single priority, as given in the following example.

Example 3.3. Recall P_2 and $\mathcal{A}_2$ in Example 2.4. Then the output of the translation (with some simplification) is: $(\mathcal{E}, \texttt{input}\ 0\ (g\ n))$ where

$$
\begin{array}{l}
\texttt{repeat}\ f\ x =_\mu (x \leq 0 \wedge \texttt{fin})\\
\qquad\qquad \vee (x > 0 \wedge (f\,x\,(\texttt{repeat}\ f) \wedge \texttt{repeat}\ f\ (x-1)))\\
\texttt{fin} =_\mu \texttt{true}\\
\texttt{sub}\ x\ y\ k =_\mu k\ (y - x)\\
g\ z\ x =_\mu \texttt{repeat}\ (\texttt{sub}\ x)\ z\\
\texttt{input}\ x\ k =_\mu (k\ x) \vee (\texttt{input}\ (x+1)\ k).
\end{array}
$$

Since $\mathcal{A}_2$ has only a single priority 1, we could avoid replications of arguments. The resulting $\mathrm{HFL_Z}$ formula mirrors the structure of $\mathrm{HORS_Z}$ P_2. Notice that angelic and demonic non-determinisms (denoted by a and d) have been replaced by $\vee$ and $\wedge$ respectively. □

[5] In fact, by specializing g with the second argument h, we obtain P_1; we have deliberately chosen such an example, to make it easy to understand how the translation works for higher-order functions.

The following theorem states the soundness and completeness of the reduction.

Theorem 3.4. *Let P be a $\mathrm{HORS_Z}$ and $\mathcal{A}$ be an APT that share the same ranked alphabet. Then, $\mathbf{Tree}(P)$ is accepted by $\mathcal{A}$ if and only if $\models \Phi_{P,\mathcal{A}}$.*

Since the proof of the theorem above is rather involved, we defer the proof to the longer version [36] here we explain only the overall structure of the proof. For the sake of clarity of the proof, we consider an intermediate problem which we call a *call sequence game*, and decompose the reduction into two steps, one from the $\mathrm{HORS_Z}$ model checking problem to the call sequence game, and the other from the call sequence game to the $\mathrm{HFL_Z}$ validity checking problem.

The call sequence game is an extension of the call sequence analysis problem considered in [22]. We consider a program (represented also as a $\mathrm{HORS_Z}$) with two kinds of branches and and or, where each recursive function is associated with a priority. During an execution of the program, when an or branch (resp. an and branch) is encountered, Player (resp. Opponent) picks one of the branches. An infinite play (i.e., an infinite execution sequence of the program) is won by Player if the largest priority occurring infinitely often in the (unique) infinite chain of recursive calls in the play is even. Player wins the call sequence game if she has a winning strategy; see [36] for the precise definition. The first translation converts an instance of the $\mathrm{HORS_Z}$ model checking problem $(P, \mathcal{A})$ to a call-sequence game, so that $\mathbf{Tree}(P) \in \mathcal{L}(\mathcal{A})$ if and only if Player wins the game. The second translation then converts the game to a $\mathrm{HFL_Z}$ formula, so that Player wins the game if and only if the formula is valid.

We prove soundness and completeness of both steps of the reductions in [36]. The correctness proof of the first reduction generalizes arguments used in decidability proofs of HORS model checking [20, 30], whereas that of the second reduction generalizes arguments used in the proof of the correctness of the translation from HORS to HFL [19].

4 Validity Checking of $\mathrm{HFL_Z}$ Formulas

We have so far presented a general reduction from APT model checking problems (which subsume linear-time/branching time model checking such as LTL/CTL/CTL* model checking) for higher-order programs (represented as $\mathrm{HORS_Z}$) to $\mathrm{HFL_Z}$ validity checking problems. The $\mathrm{HFL_Z}$ formula obtained by the reduction can be considered a kind of verification condition for the original program verification problem. We discuss how to prove the $\mathrm{HFL_Z}$ formulas interactively by using Coq in Section 4.1, and then sketch an automated approach briefly in Section 4.2. The full investigation of the latter approach is left for future work.

4.1 Interactive Validity Checking Using Coq

For the sake of simplicity, we consider here $\mathrm{HFL}_{\mathrm{N}}$ (HFL extended with natural numbers) instead of $\mathrm{HFL}_{\mathrm{Z}}$. We first describe our formalization of $\mathrm{HFL}_{\mathrm{N}}$ in Coq in Section 4.1.1 and demonstrate an example of a proof. We then describe lemmas and tactics prepared to enhance automation in Section 4.1.2. Some familiarity with Coq is assumed in this subsection.

4.1.1 Formalization of $\mathrm{HFL}_{\mathrm{N}}$ Formulas in Coq

As a running example, let us consider (the HES representation of) the $\mathrm{HFL}_{\mathrm{N}}$ formula $(\mathcal{E}, \mathtt{input}\ 0\ (g\ n))$ obtained in Example 3.3.
The goal is to prove that the formula $\mathtt{input}\ 0\ (g\ n)$ is valid for every natural number n.

In order to avoid the clumsy issue of representing variable bindings in Coq, we represent the *semantics* of the above formula in Coq, rather than the syntax. The definitions below correspond to those of $\mathcal{D}_\tau$ in Section 2. They are not specific to the above goal, and can be used as a library for proving any HFL formulas,

```
(* syntax of simple types:
   "arint t" and "ar t1 t2" represent
   nat->t and t1->t2 respectively *)
Inductive ty: Set :=
    o: ty
  | arint: ty -> ty
  | ar: ty -> ty -> ty.

(* definition of semantic domains.
   "dom t" corresponds to D_t,
   minus the monotonicity condition *)
Fixpoint dom (t:ty): Type :=
  match t with
     o => Prop
   | arint t' => nat -> dom t'
   | ar t1 t2 => (dom t1) -> (dom t2)
  end.
```

Here, we use `Prop` as the semantic domain $\mathcal{D}_\bullet = \{\mathtt{false}, \mathtt{true}\}$. and represent `true` as a proposition `True`.

Above, we have omitted the monotonicity condition, which is separately defined by induction on simple types, as follows.

```
Fixpoint ord (t:ty) {struct t}:
                    dom t -> dom t -> Prop :=
  match t with
    o => fun x: dom o => fun y: dom o => (x -> y)
  | arint t' =>
     fun x: dom (arint t') =>
     fun y: dom (arint t') =>
          forall z:nat, ord t' (x z) (y z)
  | ar t1 t2 =>
      fun x: dom (ar t1 t2) =>
      fun y: dom (ar t1 t2) =>
       forall z w:dom t1,
        ord t1 z z -> ord t1 w w ->
        ord t1 z w -> ord t2 (x z) (y w)
  end.

Definition mono (t: ty) (f:dom t) :=
    ord t f f.
```

Here, $\mathtt{ord}\ \tau$, which is defined by induction on τ, corresponds to $\sqsubseteq_\tau$ in Section 2,[6] and the monotonicity condition on f is expressed as the reflexivity condition $\mathtt{ord}\ \tau\ f\ f$.

We can then formalize the goal formula, as follows.

```
Definition fin := True.
Definition sub :=
  fun x: nat => fun y:nat =>
  fun k:nat->Prop => k(y-x).
Definition repeat_t :=
 (* type of repeat: (nat->(nat->o)->o)->nat->o *)
    ar (arint (ar (arint o) o)) (arint o).
Definition repeat_gen :=
  fun repeat: dom repeat_t =>
  fun f:dom (arint (ar (arint o) o)) =>fun x:nat=>
    (x<=0 /\ fin)
    \/ (x>0 /\ f x (repeat f) /\ repeat f (x-1)).
Definition input_t := arint (ar (arint o) o).
Definition input_gen :=
  fun input: dom (arint (ar (arint o) o)) =>
   fun x:nat => fun k: nat->Prop =>
     k x \/ input (x+1) k.
Theorem th_repeat:
 forall repeat: dom repeat_t,
 forall FPrepeat:
    (* repeat is a fixpoint of repeat_gen *)
  (forall f:dom (arint (ar (arint o) o)),
   forall x:nat,
    mono (arint (ar (arint o) o)) f ->
    repeat f x <-> repeat_gen repeat f x),
 forall input: dom input_t,
 forall FPinput:
    (* input is a fixpoint of input_gen *)
   (forall x:nat, forall k:nat->Prop,
    input x k <-> input_gen input x k),
 forall n:nat,
   input 0 (fun x:nat => repeat (sub x) n).

Proof.
(* this part should be filled by a user *)
...
Qed.
```

[6] Note, however, that since `dom t` may be inhabited by non-monotonic functions, "$\mathtt{ord}\ \tau$" is not reflexive.

Here, the definitions of `fin` and `sub` directly correspond to the equations on `fin` and `sub`. The (semantics of the) formulas `repeat` and `input` cannot be directly defined in Coq, as they are defined by using the least fixpoint operator μ. Instead, we define the functions `repeat_gen` and `input_gen`, and assume, in the statement of the theorem, that `repeat` and `input` are fixpoints of `repeat_gen` and `input_gen` respectively.

Note that except the proof (the part "..."), all the above script can be *automatically* generated from the given HORS_Z model checking problem, based on the reduction in Section 3 (like `Why3`[11], but *without* any invariant annotations). By filling the part "...", the task of HORS_Z model checking is completed. A proof script for the theorem above is found in [36]

Remark 1. In the theorem above, we have assumed that `input` is *a* fixpoint of `input_gen`. This is sufficient since `input` is the *least* fixpoint of `input_gen`, and the goal formula ("`input 0 (fun x:nat => repeat (sub x) n)`") contains only a postive occurrence of `input`; if the goal is provable with the assumption that `input` is *the least* fixpoint, then it should also be provable with the (superficially) weaker assumption that `input` is *a* fixpoint. If `input` were defined as the *greatest* fixpoint, we should add the following assumption, so that co-inductive reasoning can be used for `input`.

```
forall GFPinput:
  ((* for any postfixpoint z of input_gen *)
   forall z: dom input_t,
   mono input_t z ->
   ord input_t z (input_gen z) ->
   (* input is greater than z *)
   ord input_t z input),
...
```

The above definition allows us to prove a formula of the form $C[\texttt{input}]$ (where C is a context in which `input` occurs in a positive position) by proving instead $C[z]$ for *a* postfixpoint z of `input_gen`.

Remark 2. Notice that in the assumption `FPrepeat` of the theorem above, we require that the argument `f` is monotonic. Without the requirement, the assumption can be unsound. As a simpler example, consider the fixpoint equation $\texttt{loop}\, f =_\mu f(\texttt{loop}\, f)$. One may be tempted to assume

```
forall f:Prop->Prop, loop f <-> f(loop f)
```

in Coq. For $\texttt{f} = \lambda x.\neg x$, however, the assumption would imply `loop f <-> ¬(loop f)`, which is unsound in the presence of the law of the excluded middle. We, therefore, need to impose the condition that `f` is monotonic.

Remark 3. *Another subtlety is involved in the treatment of mutually recursive fixpoint equations. For example, consider* $(X_1 =_\nu X_1 \wedge X_2; X_2 =_\nu X_1 \wedge X_2)$*; according to the semantics of HFL formulas,* X_1 *and* X_2 *are equivalent to* `true`*. One may be tempted to prepare the following four assumptions.*

```
X1 <-> X1 /\ X2.
X2 <-> X1 /\ X2.
forall X1':Prop, ord o X1' (X1' /\ X2)
                            -> ord o X1' X1.
forall X2':Prop, ord o X2' (X1 /\ X2')
                            -> ord o X2' X2.
```

They assume that X_1 *and* X_2 *(written as* `X1` *and* `X2`*) are the greatest fixpoints of* $\lambda X_1.X_1 \wedge X_2$ *and* $\lambda X_2.X_1 \wedge X_2$ *respectively. They are, however, not strong enough to allow us to deduce* $X_1 = X_2 =$ `true`*, as* $X_1 = X_2 =$ `false` *also satisfies the assumptions above. To correctly encode* $(X_1 =_\nu X_1 \wedge X_2; X_2 =_\nu X_1 \wedge X_2)$*, we need to parameterize* X_2 *with* X_1*, like:* $(X_1 =_\nu X_1 \wedge X_2 X_1; X_2 =_\nu \lambda x_1.x_1 \wedge X_2 x_1)$ *and then define* X_2 *and* X_1 *in this order in Coq.*

Our translator (from HORS_Z model checking problems to Coq representation of HFL_Z formulas) takes the points discussed above into account.

4.1.2 Lemmas and Tactics for Semi-automation

With only plain definitions of the semantics of HFL_N formulas above, proofs of their validity are often too long and unmanageable. One of the main sources of the problem is the need to reason about the monotonicity of functions almost everywhere (recall Remark 2). For example, in the Coq script above, the condition:

```
mono (arint (ar (arint o) o)) f
```

occurs in the assumption `FPrepeat` of the main theorem; thus we need to prove this condition every time we use the assumption that `repeat` is a fixpoint of `repeat_gen`.

To address the problem above, we have prepared lemmas and tactics specialized for HFL_N, which consist of:

(i) general lemmas and tactics, prepared as a part of our Coq library for HFL_N, which can be commonly used for proving HFL_N formulas, and
(ii) instance-wise tactics, generated for each instance of the HFL_N validity checking problem.

General Lemmas and Tactics for HFL$_\text{N}$ The first group includes lemmas stating standard properties such as the transitivity of the `ord`:

```
Lemma ord_trans:
  forall t:ty, forall x y z: dom t,
    ord t x y -> ord t y z -> ord t x z.
```

We have also prepared a tactic called `hord` for solving goals of the form `ord ty s t`. This tactic first tries to solve a goal using the transitivity of `ord`. If it fails, it expands the definition of `ord` one step and tries auto. For example, the goal "`ord ft f f`" where

```
ft := ar (ar (ar o o) o) o
f := fun h : dom (ar (ar o o) o)
```

Table 1. The result of preliminary experiments

problems	w/o tactics	with tactics
ex2_3	27	15
repeat	73	22
fib	19	4
mc	30	4
sum	11	2
sum2	14	4
sum3	22	5
zeros	30	16
ho_nontermination	88	20

```
=> h (fun x : dom o => x)
```

can automatically be proved by `hord`.

Instance-wise Tactics The main reason why we need instance-wise tactics is that how to manipulate (higher-order) predicate variables depends on how they are defined in Coq. For instance, in the example in Section 4.1.1, `sub` is directly defined as a Coq function, whereas `repeat` and `input` are indirectly defined by using `FPrepeat` and `FPinput`. Thus, the former can be reduced by using a built-in conversion tactic in Coq, whereas the latter ones must use the assumptions `FPrepeat` and `FPinput`. We have thus prepared a tactic `hred` for unfolding predicate variables, so that the tactic can be used irrespectively of whether they are defined directly or indirectly (using `FPxx`).

Among others, we have also prepared a tactic `hauto`; this is analogous to the standard Coq tactic `auto`, but automatically tries HFL tactics like `hord` and `hred` explained above. These instance-wise tactics helped a lot in the semi-automation of proofs.

Preliminary Evaluation We have implemented two translators: one based on the translation in Section 3, and the other based on the translation specialized for may/must-reachability problems, as described in [22]. Both translators produce a Coq representation of the output of the respective translation, along with instance-wise tactics as described above.

To evaluate the effectiveness of tactics, we have interactively proved the Coq goals produced by the translators with and without using our tactics, and compared the size of proofs. The result is summarized in Table 1. In the table, the columns "w/o tactics" and "with tactics" respectively show the numbers of lines of proofs that are written without and with our tactics. In the table, ex2_3 and repeat are the outputs of the translator based on Section 3, and the others are those of the other translator. We can observe that the proofs using our tactics are significantly shorter. For example, for `mc`, which asserts the correctness of McCurthy's 91 function, the proof with tactics essentially applies just an induction.

4.2 Towards Automated $\mathrm{HFL}_{\mathrm{Z}}$ Validity Checking

To decide the validity of an $\mathrm{HFL}_{\mathrm{Z}}$ formula φ automatically (in a sound but incomplete manner; note that the incompleteness is inevitable, as validity checking is undecidable), we prepare the negation $\overline{\varphi}$ of φ (which is obtained by taking de Morgan dual of φ), and run procedures for proving φ and $\overline{\varphi}$ concurrently. To prove φ (or $\overline{\varphi}$), we can extend and apply the previous techniques for automated verification of functional programs based on higher-order model checking. For a ν-only formula φ, we can apply predicate abstraction [21, 24] to obtain a pure HFL formula φ' as an underapproximation of φ, and invoke a pure HFL model checker (which has already been implemented, based on the type-based characterization of HFL model checking [19]). To deal with a formula containing μ, we can extend the technique for proving termination [25, 28], to replace each μ-formula with a ν-formula, and then apply the method above for ν-only formulas. For example, consider a formula $\varphi \equiv \mu X.\lambda y.(y = 0 \vee X(y-1))$. To prove that $\models \varphi\, n$ holds for every $n \geq 0$, we transform it to $\varphi' \equiv \nu X'.\lambda y'.\lambda y.((0 \leq y < y') \wedge (y = 0 \vee X'\, y\, (y-1)))$ and prove that $\varphi'\,(n+1)\,n$ is valid for every $n \geq 0$ (which is a sufficient condition for $\varphi\, n$). Here, we record the argument of the previous call of X in the auxiliary argument y', and check that the argument of X in the original formula strictly decreases with respect to a well-founded relation.

Example 4.1. Consider the HES $(\mathcal{E}', f_{q_0,1}\,0)$ obtained in Example 3.1, where $\mathcal{E}'$ is:

$$
\begin{aligned}
g_{q_2,2} &=_\nu \lambda x.(x = 0 \wedge g_{q_2,2}\,0) \vee (x \neq 0 \wedge g_{q_1,1}\,(x-1));\\
f_{q_0,1} &=_\mu \lambda x.(f_{q_0,1}\,(x+1)) \vee (g_{q_0,1}\,x);\\
g_{q_0,1} &=_\mu \lambda x.(x = 0 \wedge g_{q_0,1}\,0) \vee (x \neq 0 \wedge g_{q_1,1}(x-1));\\
g_{q_1,1} &=_\mu \lambda x.(x = 0 \wedge g_{q_2,2}\,0) \vee (x \neq 0 \wedge g_{q_1,1}\,(x-1)).
\end{aligned}
$$

Using the well-founded orders $\prec_g$ and $\prec_f$ defined by: $x \prec_g x' \Leftrightarrow 0 \leq x < x'$ and $x \prec_f x' \Leftrightarrow (x = 0 \wedge x' = -1) \vee (x = 1 \wedge x' = 0)$,[7] we can transform the HES to $(\mathcal{E}'', f'_{q_0,q}\,(-1)\,0)$, where $\mathcal{E}''$ consists of the following ν-only equations.

$$
\begin{aligned}
g_{q_2,2} &=_\nu \lambda x.(x = 0 \wedge g_{q_2,2}\,0) \vee (x \neq 0 \wedge g'_{q_1,1}\,x\,(x-1));\\
f'_{q_0,1} &=_\nu \lambda x'.\lambda x.((x = 0 \wedge x' = -1) \vee (x = 1 \wedge x' = 0))\\
&\qquad \wedge (f'_{q_0,1}\,x\,(x+1) \vee g'_{q_0,1}\,(x+1)\,x);\\
g'_{q_0,1} &=_\nu \lambda x'.\lambda x.0 \leq x < x'\\
&\qquad \wedge ((x = 0 \wedge g'_{q_0,1}\,x\,0) \vee (x \neq 0 \wedge g_{q_1,1}\,x\,(x-1)));\\
g'_{q_1,1} &=_\nu \lambda x'.\lambda x.0 \leq x < x'\\
&\qquad \wedge ((x = 0 \wedge g_{q_2,2}\,0) \vee (x \neq 0 \wedge g'_{q_1,1}\,x\,(x-1))).
\end{aligned}
$$

We can then apply predicate abstraction [21] to underapproximate the formula and prove it true. For example, the argument of $g_{q_2,2}$ can be abstracted using the predicate $x = 0$, and obtain

$$g'_{q_2,2} =_\nu \lambda b_{x=0}.(b_{x=0} \wedge g'_{q_2,2}\,b_{x=0}) \vee \mathtt{false}.$$

[7] Such well-founded orders can be found in a counterexample-guided manner [25, 28].

Here, $b_{x=0}$ is a boolean argument that represents whether the original argument x satisfies $x = 0$. Thus, $g_{q_2,2}\,0$ is underapproximated by $g'_{q_2,2}\,\texttt{true}$, which is equivalent to `true`. □

5 Related Work

As already discussed in Section 1, there have been active studies on temporal verification of higher-order programs [12, 14, 18, 22, 23, 26, 28, 29, 35], but they were either limited to finite-data programs or linear-time properties. The present work on branching time verification is an extension of our previous work on reductions from linear-time temporal property verification to HFL_Z validity checking [22]. The extension is non-trivial; in particular, the correctness proof requires more sophisticated arguments. Thanks to the generalization, the correctness of our reduction now subsumes the decidability of HORS model checking. Actually, some of the arguments in our correctness proof can be seen as a generalization of the argument in Kobayashi and Ong's type-based proof of the decidability of HORS model checking [20]. We have also simplified the translation of [22], by avoiding the use of intersection types.

Our reduction is also closely related to Kobayashi et al.'s reduction from pure HORS model checking to pure HFL model checking [19]. The differences of our translation from the translation of [19] are: (i) the translation has been extended to deal with integers and conditional expressions, and (ii) the output is a *validity checking* problem for *modal-free* formulas, rather than a *model checking* problem for formulas containing modal operators. Intuitively, the latter has been achieved by a kind of "product construction" of a HORS_Z term and an automaton; this is why the translation of a term is parameterized only with a priority in [19], whereas it is parameterized with both a priority and a state in ours. A more fundamental difference lies in the correctness proofs. In [19], they reused the correctness of a type-based characterization of HORS model checking (which played the central role in a decidability proof of HORS model checking [20]). Since the type-based characterization is not available for HORS_Z, we have proved the correctness of the translation from scratch, by generalizing the arguments of [22]. As mentioned already, our correctness proof may be seen as a generalization of arguments used in the type-based decidability proof of HORS model checking [20]. Our syntactic translation, as well as that in [19], is closely related to the domain-theoretic model by Salvati and Walukiewicz [31], in which the semantics of a recursive function is defined by using nested least/greatest fixed-points.

Automated methods for verification of CTL or CTL* properties have been actively studied for imperative programs [4, 7–10]. We deal with larger classes of properties (arbitrary ω-regular properties expressible in APT, or equivalently, modal μ-calculus), and programs (higher-order programs with arbitrary recursion), albeit that we have so far given only the reduction to HFL_Z validity checking, leaving an implementation of an automated HFL_Z validity checker for future work.

As already mentioned, the HFL_Z formulas obtained by our reduction can be treated as verification conditions for the original program verification problems. Most of the other VCG (verification condition generation) approach to program verification [1, 11] requires invariant annotations, whereas ours does not, at the expense of using a fixpoint logic. Charguéraud [5, 6] introduced *characteristic formulas*, which are verification conditions for total correctness of functional programs, and implemented an interactive verification tool on top of Coq proof assistant. Like ours, his approach does not require invariant annotations either, although the target properties (hence also the translation from verification problems to logical formulas) are quite different. We are not aware of any previous VCG approach to *temporal property* verification of higher-order programs.

6 Conclusion

We have proposed a sound and complete reduction from HORS_Z model checking to HFL_Z validity checking, which enables verification of branching-time properties of infinite-data, higher-order programs. We have also developed a Coq library for interactively proving the HFL_Z formulas generated by the translation. Work is under way to develop an automated HFL_Z validity checker, as sketched in Section 4.

Acknowledgments

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

References

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 364–387.

[2] Stefano Berardi, Mario Coppo, Ferruccio Damiani, and Paola Giannini. 2000. Type-Based Useless-Code Elimination for Functional Programs. In *Proceedings of SAIG 2000 (LNCS)*, Vol. 1924. Springer, 172–189.

[3] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science)*, Vol. 9300. Springer, 24–51.

[4] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016.* 387–393.

[5] Arthur Charguéraud. 2010. Program verification through characteristic formulae. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 321–332.

[6] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 418–430.

[7] Byron Cook, Heidy Khlaaf, and Nir Piterman. 2015. Fairness for Infinite-State Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*. 384–398.

[8] Byron Cook, Heidy Khlaaf, and Nir Piterman. 2015. On Automation of CTL* Verification for Infinite-State Systems. In *Computer Aided Verification - 27th International Conference, CAV 2015*. 13–29.

[9] Byron Cook, Heidy Khlaaf, and Nir Piterman. 2017. Verifying Increasingly Expressive Temporal Logics for Infinite-State Systems. *J. ACM* 64, 2 (2017), 15:1–15:39.

[10] Byron Cook and Eric Koskinen. 2013. Reasoning about nondeterminism in programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 219–230.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of ESOP 2013 (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

[12] Koichi Fujima, Sohei Ito, and Naoki Kobayashi. 2013. Practical Alternating Parity Tree Automata Model Checking of Higher-Order Recursion Schemes. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Proceedings (Lecture Notes in Computer Science)*, Vol. 8301. Springer, 17–32.

[13] Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research*. LNCS, Vol. 2500. Springer.

[14] Martin Hofmann and Wei Chen. 2014. Abstract interpretation from Büchi automata. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. ACM, 51:1–51:10.

[15] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. 2002. Higher-Order Pushdown Trees Are Easy. In *FoSSaCS 2002 (LNCS)*, Vol. 2303. Springer, 205–222.

[16] Naoki Kobayashi. 2001. Type-Based Useless-Variable Elimination. *Higher-Order and Symbolic Computation* 14, 2-3 (2001), 221–260.

[17] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. 416–428.

[18] Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20.

[19] Naoki Kobayashi, Étienne Lozes, and Florian Bruse. 2017. On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 246–259.

[20] Naoki Kobayashi and C.-H. Luke Ong. 2009. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*. IEEE Computer Society Press, 179–188.

[21] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*. ACM, 222–233.

[22] Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. 2018. Higher-Order Program Verification via HFL Model Checking. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.), Vol. 10801. Springer, 711–738.

[23] Eric Koskinen and Tachio Terauchi. 2014. Local temporal reasoning. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. ACM, 59:1–59:10.

[24] Takuya Kuwahara, Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2015. Predicate Abstraction and CEGAR for Disproving Termination of Higher-Order Functional Programs. In *Proceedings of CAV 2015 (Lecture Notes in Computer Science)*, Vol. 9207. Springer, 287–303.

[25] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Proceedings of ESOP 2014 (Lecture Notes in Computer Science)*, Vol. 8410. Springer, 392–411.

[26] M. M. Lester, R. P. Neatherway, C.-H. Luke Ong, and S. J. Ramsay. 2011. Model checking liveness properties of higher-order functional programs. In *Proceedings of ML Workshop 2011*.

[27] Étienne Lozes. 2015. A Type-Directed Negation Elimination. In *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015. (EPTCS)*, Ralph Matthes and Matteo Mio (Eds.), Vol. 191. 132–142.

[28] Akihiro Murase, Tachio Terauchi, Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2016. Temporal verification of higher-order functional programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. ACM, 57–68.

[29] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768.

[30] C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science, LICS 2006, Proceedings*. IEEE Computer Society, 81–90.

[31] Sylvain Salvati and Igor Walukiewicz. 2015. A Model for Behavioural Properties of Higher-order Programs. In *Proceeding of the 24th EACSL Annual Conference on Computer Science Logic, CSL 2015*. 229–243.

[32] Moshe Y. Vardi. 1991. Verification of Concurrent Programs: The Automata-Theoretic Framework. *Ann. Pure Appl. Logic* 51, 1-2 (1991), 79–98.

[33] M. Viswanathan and R. Viswanathan. 2004. A Higher Order Modal Fixed Point Logic. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 512–528.

[34] Mitchell Wand and Igor Siveroni. 1999. Constraint Systems for Useless Variable Elimination. In *Proceedings of POPL*. 291–302.

[35] Keiichi Watanabe, Ryosuke Sato, Takeshi Tsukada, and Naoki Kobayashi. 2016. Automatically disproving fair termination of higher-order functional programs. In *Proceedings of ICFP 2016*. ACM, 243–255.

[36] Keiichi Watanabe, Takeshi Tsukada, Hiroki Oshikawa, and Naoki Kobayashi. 2019. Reduction from Branching-time Property Verification of Higher-Order Programs to HFL Validity Checking. A longer vesion, available at http://www-kb.is.s.u-tokyo.ac.jp/~koba/papers/pepm2019-full.pdf.