Automatic HFL(Z) Validity Checking for Program Verification

Naoki Kobayashi Kento Tanahashi Ryosuke Sato
The University of Tokyo
Takeshi Tsukada
Chiba University
December 12, 2022

Abstract

We propose an automated method for checking the validity of a formula of HFL(Z), a higher-order logic with fixpoint operators and integers. Combined with Kobayashi et al.’s reduction from higher-order program verification to HFL(Z) validity checking, our method yields a fully automated, uniform verification method for arbitrary temporal properties of higher-order functional programs expressible in the modal μ-calculus, including termination, non-termination, fair termination, fair non-termination, and also branching-time properties. We have implemented our method and obtained promising experimental results.

1 Introduction

Kobayashi et al. [22, 42] have shown that temporal property verification problems for higher-order functional programs can be reduced to the validity checking problem for HFL(Z). HFL(Z) is an extension of Viswanathan and Viswanathan’s higher-order fixpoint logic (HFL) [40] with integers, and the validity checking problem asks whether or not a given HFL(Z) formula (without modal operators) is valid. The reduction provides a uniform approach to the temporal property verification of higher-order functional programs. Automatic validity checkers have been implemented for the first-order fragment of HFL(Z) [19] and νHFL(Z) [15, 17], the fragment of HFL(Z) without least fixpoint operators. The former [19] enables automated verification of temporal properties of first-order programs, and the latter [15, 17] enables automated verification of safety properties of higher-order programs. This line of work provides a streamlined, general approach to automated verification of temporal properties of programs. Despite the generality of the approach, it has been reported that Kobayashi et al.’s tool [19] outperformed Cook and Koskinen’s method specialized for CTL verification [8].

Following the above line of research, we propose an automated (sound but incomplete) method of validity checking for the full fragment of HFL(Z). By combining the proposed method with the above-mentioned reduction [22, 42], we can obtain a fully automated verification method for arbitrary regular temporal properties of higher-order programs. The properties that can be verified in a uniform manner using our method include safety [34, 37, 20, 35, 30, 32, 43], termination [25], non-termination [24], fair termination [27], and fair

\footnote{Incompleteness is inevitable because the validity checking problem for HFL(Z) is undecidable in general.}
Table 1: Fixpoint logics for program verification

| Level     | µ- or υ-only          | both µ and υ          |
|-----------|------------------------|------------------------|
| first-order | Constrained Horn Clauses (CHCs) | Mu-Arithmetic [10] |
| higher-order | υHFL(Z) [13, 17] | HFL(Z) [22, 42] |

non-termination [41], for which separate methods and tools have been developed so far. Furthermore, our method can also be used for automatic verification of branching-time properties (typically expressed by formulas of CTL, CTL*, and the modal µ-calculus), which have not been supported by previous automated methods/tools for higher-order program verification, to our knowledge.

Table 1 compares HFL(Z) with other fixpoint logics studied in the context of automated program verification. Program verification by reduction to the satisfiability problem of constrained Horn clauses (CHCs) has recently been studied actively as a uniform method for automated verification of first-order programs [2]. As discussed in [19], the CHC satisfiability problem corresponds to the validity checking problem for the first-order fragment of HFL(Z) with only the greatest fixpoint operators; thus CHCs can be used to verify safety properties of first-order programs, but extensions [11, 3] are required to reason about other properties such as liveness and termination. The Mu-Arithmetic studied by Kobayashi et al. [19] allows arbitrary alternations of greatest and least fixpoint operators, and can be used for verification of arbitrary regular properties of first-order programs, but not higher-order ones. Burn et al. [6] studied a higher-order extension of CHCs, and Katsura et al. [15, 17] studied the corresponding fragment of HFL(Z) called νHFL(Z). They can be used for verification of safety properties of higher-order programs, but not arbitrary temporal properties. The automated method for HFL(Z) validity checking developed in this paper enables a uniform approach to automated verification of arbitrary regular temporal properties of higher-order programs.

Our approach to automated validity checking of HFL(Z) formulas has been inspired by the approach of Kobayashi et al. [19] for first-order HFL(Z) and that of Fedyukovich et al. [11] for termination analysis of first-order programs. We approximate a given HFL(Z) formula with a formula of νHFL(Z), the fragment of HFL(Z) without the least-fixpoint operator. We can then use existing solvers [15, 17] to prove the validity of νHFL(Z) formula. The idea of removing the least-fixpoint operator is as follows. Suppose we wish to prove that (µX.ϕ(X)) y holds for every integer y, where µX.ϕ(X) represents the least predicate such that X = ϕ(X); for example, µX.λz.X(z) is equivalent to λz.false, and µX.λz.z = 0 ∨ X(z − 1) is equivalent to λz.z ≥ 0. By a standard property of least fixpoints, µX.ϕ(X) can be under-approximated by a formula of the form ϕ(e(λz.false)) ≡ ϕ(⋯ϕ(λz.false)⋯), where e is an expression denoting a non-negative integer. (The formula ϕ(e(λz.false)) is actually represented by using the greatest fixpoint operator, as discussed later.) We then use an existing νHFL(Z) validity checker to check the validity of (ϕ(e(λz.false))) y. If (ϕ(e(λz.false))) y is valid, then we can conclude that the original formula (µX.ϕ(X)) y is also valid. Otherwise, we increase the value of e to improve the precision and run the νHFL(Z) validity checker again. Following Kobayashi et al.’s work on the first-order case [19], we consider as e an expression of the form c_0 + c_1|x_1| + ⋯ + c_k|x_k|, where x_1, ⋯, x_k are the integer variables in scope, and gradually increase the coefficients c_0, ⋯, c_k.

The new challenge in this paper for dealing with the higher-order case is how to incorporate the values of higher-order variables into e. To this end, for each function argument,
we add an extra integer argument that represents information about the function argument (thus, a predicate of the form \( \lambda f.\varphi \) would be transformed to \( \lambda (v_f, f).\varphi' \), where \( v_f \) is the extra integer argument that represents information about \( f \), and used in \( \varphi' \) to compute the value of \( e \) above). The idea of adding extra arguments has been inspired by the work of Unno et al. \[39\] on relatively complete verification of safety properties of higher-order functional programs, but we have devised a different, more systematic method for inserting extra arguments. To avoid the insertion of unnecessary extra arguments, we also propose a type-based static analysis to estimate necessary extra arguments.

The contributions of this paper are summarized as follows.

1. An extension of Kobayashi et al.’s method for the first-order HFL(Z) \[19\], to obtain an automated validity checking method for full HFL(Z).
2. A method of adding extra arguments for higher-order arguments, to improve the precision.
3. An optimization to avoid the insertion of unnecessary extra arguments.
4. A theoretical characterization of the power of our method (Section 4). We compare our method with previous popular methods for proving termination, such as those using lexicographic linear ranking functions and disjunctive well-founded relations.
5. An implementation and experiments on the proposed methods above (Section 5). According to the experiments, our tool outperformed previous verification tools specialized for verification of termination \[20\], non-termination \[21\], fair termination \[27\], and fair non-termination \[41\]. We have also confirmed that our tool can verify properties of higher-order programs that were not supported by previous automated tools, including branching-time properties.

The rest of this paper is structured as follows. Section 2 reviews HFL(Z) and its connection to program verification. Section 3 describes our method for HFL(Z) validity checking. Section 4 gives some theoretical characterization of the power of our method by comparing it with previous methods for proving termination and liveness properties. Section 5 reports experimental results. Section 6 discusses related work and Section 7 concludes the paper.

## 2 Preliminaries

This section reviews HFL(Z) and its application to program verification. HFL(Z) is an extension of Viswanathan and Viswanathan’s higher-order fixpoint logic (HFL) \[40\] with integers.

### 2.1 HFL(Z)

The set of types, ranged over by \( \kappa \), is given by:

\[
\kappa \text{ (types)} ::= \text{Int} \mid \tau \quad \tau \text{ (predicate types)} ::= * \mid \kappa \to \tau.
\]

Here, * is the type of propositions, and Int is the type of integers. A predicate type \( \tau \) is of the form \( \kappa_1 \to \cdots \to \kappa_k \to * \), which describes \( k \)-ary (possibly higher-order) predicates.
Here, and use it only informally in this paper.

\( \nu x \) define the order and arity of \( (HES) \) \[18, 22\]. We omit the formal definition of the HES representation on values of types \( \alpha x.\lambda y.\phi t \) ions. For example, we call \( \mu x \) predicates that takes an integer and a predicate on integers as arguments. For a type \( \kappa \), we define the order and arity of \( \kappa \), written \( \text{ord}(\kappa) \) and \( \text{ar}(\kappa) \) respectively, by:

\[
\begin{align*}
\text{ord}(\text{Int}) &= \text{ord}(\ast) = 0 & \text{ord}(\kappa \rightarrow \tau) &= \max(\text{ord}(\tau), \text{ord}(\kappa) + 1) \\
\text{ar}(\text{Int}) &= \text{ar}(\ast) = 0 & \text{ar}(\kappa \rightarrow \tau) &= \text{ar}(\tau) + 1.
\end{align*}
\]

The syntax of HFL(Z) formulas is given as follows.

\[
\begin{align*}
\varphi \text{ (formulas)} &::= x \mid \varphi_1 \lor \varphi_2 \mid \varphi_1 \land \varphi_2 \\
&\mid \mu x^\tau.\varphi \mid \nu x^\tau.\varphi \quad \text{(fixpoint operators)} \\
&\mid \varphi_1 \varphi_2 \mid \lambda x^\kappa.\varphi \quad \text{(\( \lambda \)-abstractions and applications)} \\
&\mid \varphi e \mid e_1 \geq e_2 \quad \text{(extension with integers)}
\end{align*}
\]

\( e \) (integer expressions) \( ::= n \mid x \mid e_1 + e_2 \mid e_1 \times e_2 \).

Here, \( x \) and \( n \) are metavariables for variables and integers respectively. The formulas \( \mu x^\tau.\varphi \) and \( \nu x^\tau.\varphi \) respectively denote the least and greatest predicates \( x \) such that \( x = \varphi \). For example, \( \mu x^\ast.x \) and \( \nu x^\ast.x \) are equivalent to \( \text{false} \) (which can be expressed as \( 0 \geq 1 \)) and \( \text{true} \) (which can be expressed as \( 0 \geq 0 \)) respectively. The variable \( x \) is bound in \( \mu x^\tau.\varphi \), \( \nu x^\tau.\varphi \), and \( \lambda x^\tau.\varphi \). As usual, we implicitly assume \( \alpha \)-renaming of bound variables. We write \( [\varphi_1/x]\varphi_2 \) for the capture-avoiding substitution of \( \varphi_1 \) for all the free occurrences of \( x \) in \( \varphi_2 \). We often omit the type annotation. Henceforth, we often use shorthand notations like \( e_1 = e_2 \) (for \( e_1 \geq e_2 \land e_2 \geq e_1 \)) and \( e_1 - e_2 \) (for \( e_1 + (-1) \times e_2 \)) and treat them as if they were primitives.

We consider only formulas well-typed under the simple type system given in Fig. 1. In the figure, \( \Gamma \) denotes a type environment of the form \( x_1: \kappa_1, \ldots, x_k: \kappa_k \), which is considered a function that maps \( x_i \) to \( \kappa_i \) for \( i \in \{1, \ldots, k\} \). For example, \( \mu x^{\text{Int} \rightarrow \ast}.x \) 1 is rejected as ill-typed (since the fixpoint variable \( x \) and \( x \) 1 have different types, violating T-Mu)).

**Notation 1.** For readability, we sometimes represent fixpoint formulas by using fixpoint equations. For example, we call \( \alpha x.\lambda y.\varphi \) (where \( \alpha = \mu \) or \( \nu \)) "the predicate \( x \) defined by the equation \( xy = \alpha \varphi \)." The latter presentation of fixpoint formulas is in general called hierarchical equation systems (HES) \[18, 22\]. We omit the formal definition of the HES representation and use it only informally in this paper. \( \Box \)
We review the formal semantics of HFL($\mathbb{Z}$) formulas. For each simple type $\kappa$, we define the partially ordered set $[\kappa] = ([\kappa], \sqsubseteq_\kappa)$ by:

$$[\text{Int}] = \mathbb{Z} \quad \sqsubseteq_{\text{Int}} = \{(n, n) \mid n \in \mathbb{Z}\}$$
$$[\varnothing] = \{(\varnothing, \varnothing), (\varnothing, \top), (\top, \varnothing), (\top, \top)\}$$

$$(\kappa \to \tau) = \{f \in (\kappa) \to (\tau) \mid \forall x, y \in (\kappa). x \sqsubseteq_\kappa y \Rightarrow f(x) \sqsubseteq_\tau f(y)\}$$

$$\sqsubseteq_{\kappa \to \tau} = \{(f, g) \in (\kappa \to \tau) \times (\kappa \to \tau) \mid \forall x \in (\kappa). f(x) \sqsubseteq_\tau g(x)\}$$

Here, $\mathbb{Z}$ denotes the set of integers. For each $\tau$, $[\tau]$ (but not $[\text{Int}]$) forms a complete lattice. We write $\sqsubseteq_\tau (\top, \top)$ for the least (greatest, resp.) element of $[\tau]$, and $\sqcap_\tau (\sqcup_\tau$, resp.) for the greatest lower bound (least upper bound, resp.) operation with respect to $\sqsubseteq_\tau$. We also define the least and greatest fixpoint operators $\text{LFP}_\tau, \text{GFP}_\tau \in ([\tau \to \tau] \to \tau)$ by:

$$\text{LFP}_\tau(f) = \sqcap\{g \in ([\tau]) \mid f(g) \sqsubseteq_\tau g\} \quad \text{GFP}_\tau(f) = \sqcup\{g \in ([\tau]) \mid g \sqsubseteq_\tau f(g)\}$$

Note that $\text{LFP}_\tau$ and $\text{GFP}_\tau$ are well-defined, since every element of $([\tau \to \tau])$ is a monotonic function over a complete lattice. By Tarski’s fixpoint theorem, $\text{LFP}_\tau(f)$ and $\text{GFP}_\tau(f)$ coincide with the least and greatest fixpoint of $f$, respectively.

For a simple type environment $\Gamma$, we write $[\Gamma]$ for the set of maps $\rho$ such that $\text{dom}(\rho) = \text{dom}(\Gamma)$ and $\rho(x) \in ([\Gamma(x)])$ for each $x \in \text{dom}(\rho)$.

For each valid type judgment $\Gamma \vdash_\tau \varphi : \kappa$, its semantics $[\Gamma \vdash_\tau \varphi : \kappa] \in ([\Gamma] \to ([\kappa])$ is defined by:

$$[\Gamma, x : \kappa \vdash_\tau x : \kappa] (\rho) = \rho(x)$$

$$[\Gamma \vdash_\tau \varphi_1 \lor \varphi_2 : \star] \rho = [\Gamma \vdash_\tau \varphi_1 : \star] \rho \sqcup_\tau [\Gamma \vdash_\tau \varphi_2 : \star] \rho$$

$$[\Gamma \vdash_\tau \varphi_1 \land \varphi_2 : \star] \rho = [\Gamma \vdash_\tau \varphi_1 : \star] \rho \sqcap_\tau [\Gamma \vdash_\tau \varphi_2 : \star] \rho$$

$$[\Gamma \vdash_\tau \mu x.\varphi : \tau] \rho = \text{LFP}_\tau(\lambda v \in ([\tau]). \{[\Gamma, \varphi \vdash_\tau \tau] \rho (x \mapsto v)\})$$

$$[\Gamma \vdash_\tau \nu x.\varphi : \tau] \rho = \text{GFP}_\tau(\lambda v \in ([\tau]). \{[\Gamma, \varphi \vdash_\tau \tau] \rho (x \mapsto v)\})$$

$$[\Gamma \vdash_\tau \lambda x.\varphi : \kappa \to \tau] \rho = \lambda w \in ([\kappa]). \{[\Gamma, \varphi \vdash_\tau \tau] \rho (x \mapsto w)\}$$

$$[\Gamma \vdash_\tau \varphi_1 \varphi_2 : \tau] \rho = [\Gamma \vdash_\tau \varphi_1 : \tau_2 \to \tau] \rho ([\Gamma \vdash_\tau \varphi_2 : \tau_2] \rho)$$

$$[\Gamma \vdash_\tau e : \tau] \rho = [\Gamma \vdash_\tau \varphi_1 : \text{Int} \to \tau] \rho ([\Gamma \vdash e : \text{Int}] \rho)$$

For a closed formula $\varphi$ of type $\tau$, we often just write $[\varphi]$ for $[0 \vdash_\tau \varphi : \tau]$. We write $\varphi_1 \equiv_{\Gamma, \tau} \varphi_2$ when $[\Gamma \vdash_\tau \varphi_1 : \tau] = [\Gamma \vdash_\tau \varphi_2 : \tau]$, and often omit the subscripts $\Gamma$ and $\tau$. Note in particular that the following laws hold (under an appropriate assumption on types): (i) $\alpha \varphi \equiv (\alpha \cdot \varphi)$, (ii) $\beta\varphi \equiv (\varphi_2/x_\varphi) (\beta\text{-equality})$, and (iii) $\varphi \equiv\lambda x.\varphi x$ ($\eta\text{-equality}$).

**Example 1** Consider the formula $\mu x.\text{Int} \to \cdot \lambda y. y = 0 \lor x(y - 1)$, which denotes the least predicate on integers that satisfies the equivalence $x \equiv \lambda y. y = 0 \lor x(y - 1)$. To understand what the formula means, let us expand the equality as follows.

$$x \equiv \lambda y. y = 0 \lor x(y - 1)$$

$$\equiv \lambda y. y = 0 \lor (\lambda y. y = 0 \lor x(y - 1))(y - 1) \quad (\text{expand } x)$$

$$\equiv \lambda y. y = 0 \lor x(y - 1) = 0 \lor x(y - 2) \quad (\beta\text{-reduction})$$

$$\equiv \lambda y. y = 0 \lor x(y - 1) = 0 \lor (\lambda y. y = 0 \lor x(y - 1))(y - 2) \quad (\text{expand } x)$$

$$\equiv \lambda y. y = 0 \lor x(y - 1) = 0 \lor x(y - 2) = 0 \lor x(y - 3) \quad (\beta\text{-reduction})$$

$$\equiv \ldots.$$
Thus, the formula $\mu x.\lambda y.y = 0 \lor x(y - 1)$ is equivalent to $\lambda y.y \geq 0$. □

Example 2 Consider the formula $\nu x.\lambda y.p(0) \land x(\lambda y.p(y + 1))$, which represents the greatest predicate $x$ such that $x(p) \equiv p(0) \land x(\lambda y.p(y + 1))$ for every unary predicate $p$ on integers. We can expand the equality as follows.

$$
\begin{align*}
x(p) & \equiv p(0) \land (\lambda y.p(0) \land x(\lambda y.p(y + 1)))(\lambda y.p(y + 1)) \\
& \equiv p(0) \land p(1) \land x(\lambda y.p(y + 2)) \equiv p(0) \land p(1) \land p(2) \land x(\lambda y.p(y + 3)) \equiv \cdots.
\end{align*}
$$

Thus, $\nu x.\lambda y.p(0) \land x(\lambda y.p(y + 1))$ is equivalent to $\lambda p.\forall y \geq 0.p(y)$. In this manner, universal quantifiers can be expressed by using the greatest fixpoint operator $\nu$ (note that $\forall y.p(y)$ can be expressed as $\forall y \geq 0.p(y) \land p(-y)$). Similarly, existential quantifiers can be expressed by using the least fixpoint operator $\mu$. Henceforth, we use quantifiers as if they were primitives. □

The validity checking problem for $\text{HFL}(\mathbb{Z})$ (or, the $\text{HFL}(\mathbb{Z})$ validity checking problem) asks whether a given (closed) $\text{HFL}(\mathbb{Z})$ formula $\varphi$ is valid (i.e. whether $[\varphi] = \top$). The problem is undecidable in general; we aim to develop an incomplete but sound method for proving or disproving the validity of $\text{HFL}(\mathbb{Z})$.

The fragment of $\text{HFL}(\mathbb{Z})$ without the least fixpoint operator $\mu$ is called $\nu\text{HFL}(\mathbb{Z})$. In this paper, we shall develop an automated method for $\text{HFL}(\mathbb{Z})$ validity checking, by reduction to $\nu\text{HFL}(\mathbb{Z})$ validity checking; for $\nu\text{HFL}(\mathbb{Z})$ validity checking, a few tools are available [6, 15, 17].

2.2 Applications of $\text{HFL}(\mathbb{Z})$ to Program Verification

Watanabe et al. [12] have shown that given a higher-order functional program $P$ and a formula $A$ of the modal $\mu$-calculus (or, equivalently, an alternating parity tree automaton), one can effectively construct a $\text{HFL}(\mathbb{Z})$ formula $\varphi_{P,A}$ such that the program $P$ satisfies the property described by $A$, if and only if the formula $\varphi_{P,A}$ is valid. Various temporal property verification problems (including safety, termination, CTL, LTL, CTL* verification) can thus be reduced to the $\text{HFL}(\mathbb{Z})$ validity checking problem and solved in a uniform manner. Here, we just give some examples, instead of reviewing the general reduction.

Let us consider the following OCaml program.

let rec fib x k = if x<2 then k x else fib (x-1) (fun y -> fib (x-2) (fun z -> k(y+z)))

The function $\text{fib}$ computes the Fibonacci number in the continuation-passing style. The termination of $\text{fib}$ ($\text{fun r} \rightarrow (\cdot)$) for all $x$ can be reduced to the validity of $\forall x.\text{Fib} \ x \ (\lambda r.\text{true})$, where the predicate $\text{Fib}$ is defined by:

$\text{Fib}^{\text{Int} \rightarrow (\text{Int} \rightarrow \ast \rightarrow *)} \ x \ k =_\mu (x < 2 \Rightarrow k \ x) \land (x \geq 2 \Rightarrow \text{Fib}(x - 1) \ (\lambda y.\text{Fib}(x - 2) \ (\lambda z.k(y + z))))$.

Here, $b \Rightarrow \varphi$ abbreviates $\neg b \lor \varphi$. The equation above defines $\text{Fib}$ as the least predicate that satisfies the equation (recall Notation[I]). The formula mimics the structure of the program. In particular, the parts "$x < 2 \Rightarrow k \ x$" and "$x \geq 2 \Rightarrow \cdots"$ respectively correspond to the then-part and the else-part of the function definition of $\text{fib}$.

The property "$\text{fib} \ x \ (\text{fun r} \rightarrow \text{assert}(r \geq x))$ never fails for any $x$" can be expressed by $\forall x.\text{Fib}_{\text{safe}} \ x \ (\lambda r.\ r \geq x)$, where $\text{Fib}_{\text{safe}}$ is defined by:

$\text{Fib}_{\text{safe}}^{\text{Int} \rightarrow (\text{Int} \rightarrow \ast \rightarrow *)} \ x \ k =_\nu (x < 2 \Rightarrow k \ x) \land (x \geq 2 \Rightarrow \text{Fib}_{\text{safe}}(x - 1) \ (\lambda y.\text{Fib}_{\text{safe}}(x - 2) \ (\lambda z.k(y + z))))$.

Here, we restrict the answer type of the continuation to the type unit of the unit value $\bot$, which is mapped to the type $\ast$ of propositions by the translation to $\text{HFL}(\mathbb{Z})$ validity checking.
As in the examples above, (i) the $HFL(Z)$ formula obtained by the reduction mimics the structure of the original program, and (ii) liveness properties (like termination) are expressed by using the least fixpoint operator $\mu$, and safety properties (like partial correctness) are expressed by using the greatest fixpoint operator $\nu$.

For the automated verification of first-order programs, it has been a popular approach to reduce verification problems to the satisfiability problem for Constrained Horn Clauses (CHC) [2]. Since the satisfiability problem for CHC (where data domains are restricted to integers) can be reduced to the validity checking problem for the first-order $\nu HFL(Z)$ [19], the program verification framework based on $HFL(Z)$ can be considered an extension of the CHC-based program verification framework with higher-order features and fixpoint alternations. $HFL(Z)$ can also be viewed as an extension of HoCHC (higher-order CHC) [6] with fixpoint alternations (recall Table 1).

3 Reduction from $HFL(Z)$ to $\nu HFL(Z)$

3.1 Overview of $HFL(Z)$ Validity Checking

Fig. 2 shows the overall flow of our $HFL(Z)$ validity checking method. Given a $HFL(Z)$ formula $\varphi$, we approximate $\varphi$ by a $\nu HFL(Z)$ formula $\varphi'$, by removing all the least fixpoint formulas (of the form $\mu x.\psi$). The formula $\varphi'$ is an under-approximation of $\varphi$, in the sense that if $\varphi'$ is valid, then so is $\varphi$. We then check whether $\varphi'$ is valid by using an existing validity checker for $\nu HFL(Z)$ [15, 17]. If $\varphi'$ is valid, then we can conclude that $\varphi$ is also valid. Otherwise, we refine the approximation of $\varphi$ and repeat the cycle. As the procedure in Fig. 2 can only conclude the validity of a given formula, we actually run the procedure for a given formula $\varphi$ and its negation $\neg \varphi$ (which can also be represented as a $HFL(Z)$ formula, by taking the dual of each operator) in parallel. If the procedure for $\neg \varphi$ returns “valid”, then we can conclude that $\varphi$ is invalid. Since $HFL(Z)$ validity checking is undecidable in general, the whole procedure is of course sound but incomplete: for some input, the procedure may repeat the cycle indefinitely, or the backend $\nu HFL(Z)$ validity checker [15, 17] may not terminate.

The main technical issue in the procedure sketched above is how to approximate $\mu$-formulas, which is the focus of the rest of this section. We first discuss a basic method in Section 3.2 and then discuss how to improve the precision of the approximation by adding extra arguments for higher-order predicates in Section 3.3. We then further improve the approximation by removing redundant extra arguments in Section 3.4.
3.2 Basic Method

As mentioned in Section 1, the basic idea is to under-approximate each \( \mu \)-formula \( \mu x^\tau . \varphi(x) \) (where \( \varphi \) has type \( \tau \rightarrow \tau \)) by \( \varphi^n(\bot_\tau) \triangleq \underbrace{\varphi(\varphi(\varphi(\cdots \varphi(}_{n \text{ times}}(\bot_\tau)\cdots )). \) Here, \( \bot_\tau \) is the least formula of type \( \tau \), defined by: \( \bot_\tau = \text{false} \) and \( \bot_{\kappa \rightarrow \tau} = \lambda y^\kappa . \bot_\tau \). To see that \( \varphi^n(\bot_\tau) \) is an underapproximation of \( \mu x^\tau . \varphi(x) \), recall that \( \llbracket \varphi \rrbracket \) is a monotonic function. Thus, we have:

\[
\llbracket \varphi^n(\bot_\tau) \rrbracket = \llbracket \varphi \rrbracket^n(\bot_\tau) \\
\subseteq \llbracket \varphi \rrbracket^n(\LFP(\llbracket \varphi \rrbracket)) \quad \text{(by \( \bot_\tau \subseteq \tau \) \LFP(\llbracket \varphi \rrbracket) and the monotonicity of \( \llbracket \varphi \rrbracket \))} \\
= \LFP(\llbracket \varphi \rrbracket) \quad \text{(by the definition of \( \LFP(\varphi) \))} \\
= [\mu x^\tau . \varphi(x)]
\]

Since the appropriate number \( n \) may depend on the values of free variables in \( \varphi \), we actually use the \( \nu \)-formula \( \psi n \) to represent \( \varphi^n(\bot_\tau) \), where \( \psi \) is:

\[
\nu x^{\text{Int}} \rightarrow \tau . \lambda u^{\text{Int}} . \lambda y^\kappa . u > 0 \land \varphi(x(u - 1)) \tilde{y}
\]

Here, \( \tilde{y}^\kappa \) denotes a sequence \( y_1^\kappa, \ldots, y_\ell^\kappa \) and we assume that \( \tau = \kappa_1 \rightarrow \cdots \rightarrow \kappa_\ell \rightarrow \ast \). Note that the original formula \( \mu x^\tau . \varphi x \) is \( \eta \)-equivalent to \( \mu x^\tau . \lambda y^\kappa . \varphi x \tilde{y} \); thus, the main differences of the formula \( \psi n \) from the original formula are: (i) an extra integer argument \( u \) has been added to \( x \), to count the number of iterations \( n \), and (ii) the least fixpoint operator \( \mu \) has been replaced by the greatest fixpoint operator \( \nu \). We can confirm that the formula \( \psi n \) is equivalent to \( \varphi^n(\bot_\tau) \) as follows. For \( n = 0 \), we have:

\[
\psi n \equiv (\lambda u^{\text{Int}} . \lambda y^\kappa . u > 0 \land \varphi(x(u - 1)) \tilde{y}) 0 \equiv \lambda y^\kappa . 0 > 0 \land \varphi(x(-1)) \tilde{y} \equiv \lambda y^\kappa . \text{false} \equiv \bot_\tau.
\]

and for \( n > 0 \),

\[
\psi n \equiv (\lambda u^{\text{Int}} . \lambda y^\kappa . u > 0 \land \varphi(x(u - 1)) \tilde{y}) n \quad \text{(unfolding \( \nu \))} \\
\equiv \lambda y^\kappa . n > 0 \land \varphi(x(n - 1)) \tilde{y} \quad \text{(\( \beta \)-equality)} \\
\equiv \lambda y^\kappa . \varphi(x(n - 1)) \tilde{y} \quad \text{(by assumption \( n > 0 \))} \\
\equiv \lambda y^\kappa . \varphi(\varphi^{n-1}(\bot_\tau)) \tilde{y} \quad \text{(by induction on \( n \))} \\
\equiv \varphi^n(\bot_\tau) \quad \text{(by \( \eta \)-equality)}
\]

When \( \varphi \) contains free integer variables \( w_1, \ldots, w_m \), the number \( n \) needed to properly approximate the original formula may depend on them. Thus, we actually replace the number \( n \) with an expression \( c|w_1| + \cdots + c|w_m| + d \), where \( c \) and \( d \) are some non-negative integers. Due to the monotonicity of \( \psi n \) with respect to \( n \), we can improve the precision by increasing the values of \( c \) and \( d \).

The method sketched above is a generalization of Kobayashi et al.’s method [13] for the first-order fragment of HFL(Z) (called Mu-Arithmetic) to full HFL(Z) (which was in turn a generalization the method of Fedyukovich et al. [11] for termination analysis). We give a few examples below. Example 6 highlights a subtle issue caused by the generalization to the higher-order case.

Example 3 Recall the formula \( \varphi_1 \triangleq \mu x^{\text{Int}} \rightarrow \ast . \lambda y . y = 0 \lor x(y - 1) \) in Example 2. Suppose we wish to prove the validity of \( \forall \nu . w < 0 \lor \varphi_1 w \). Based on the method sketched above, we approximate \( \varphi_1 \) with \( \psi (c|y| + d) \), where \( \psi \) is:

\[
\nu x^{\text{Int}} \rightarrow \text{Int} \rightarrow \ast . \lambda u^{\text{Int}} . \lambda y^{\text{Int}} . u > 0 \land (y = 0 \lor x(u - 1)(y - 1))
\]
The resulting formula $\forall w. w < 0 \lor \psi (c|w| + d) w$ can automatically be proved valid for $c = d = 1$, by using an existing $\nu HFL(\mathbb{Z})$ validity checker like ReTHFL [17].

To confirm the validity of $\forall w. w < 0 \lor \psi (|w| + 1) w$ manually, it suffices to observe that (the semantics of) $\forall u. \forall y. y > 0 \land u > y$ is a post-fixpoint of $\varphi_0 \triangleq \lambda x. \mu \exists x_1 + \cdots + c|x_k| + d$), where $\text{Fib}$ is defined by:

Example 5
Recall the formula $\forall u. \forall y. y > 0 \land u > y$ is a post-fixpoint of $\varphi_0 \triangleq \lambda x. \mu \exists x_1 + \cdots + c|x_k| + d$), where $\text{Fib}$ is defined by:

Remark 1
The transformation sketched above passes around extra arguments of the form $\lambda y. \varphi_{\mu u}(\forall x_1 + \cdots + c|x_k| + d)$ with

$$
\forall u. (\forall x_1 + \cdots + c|x_k| + d) \Rightarrow \varphi_{\mu u} y.
$$

For example, the formula $\lambda y. \varphi(c|y| + d) y$ above is actually replaced by:

$$
\lambda y. \forall u. u \geq c y + d \land u \geq -c y + d \Rightarrow \psi_{\mu y} u.
$$

Note that due to the monotonicity of $\psi_{\mu y}$ with respect to $u$, the replacement does not change the semantics of formulas.

Example 4
Suppose that we wish to prove the validity of $\forall w. w < 0 \lor \varphi_1 (2w)$, instead of the formula $\forall w. w < 0 \lor \varphi_1 w$, in Example 3. In this case, the approximate formula $\forall w. w < 0 \lor \psi (c|w| + d) (2w)$ is invalid when $c = d = 1$. In fact, for $w = 1$,

$$
\psi (|w| + 1) (2w) \equiv \psi 22 \equiv \psi 11 \equiv \psi 00 \equiv \text{false}.
$$

In such a case, we proceed to the approximation refinement step in Fig. 2 to increase the values of $c$ and $d$. By increasing the values of $c$ and $d$ to 2, we obtain a better approximation: $\forall w. w < 0 \lor \psi (2|w| + 2) (2w)$, which can be proved valid.

Example 5
Recall the formula $\varphi_2 \triangleq \forall x. \text{Fib} x (\lambda y. \text{true})$ in Section 2.2 (which was obtained by encoding the termination problem for Fibonacci function), where $\text{Fib}$ is defined by:

$$
\text{Fib}(x) = \begin{cases} 
\mu y. (x < 2 \Rightarrow k x) \land (x \geq 2 \Rightarrow \text{Fib}(x - 1) \lambda y. \text{Fib}(x - 2) \lambda z. k(y + z)). 
\end{cases}
$$
The formula $\varphi_2$ can be approximated by $\forall x. F_F(x,y) = (\lambda r.x \rightarrow (\lambda y.z(y + z)))$, where $F_F : \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$ is defined by:

$$F_F(x,y) = \mu_y u > 0 \land (x < 2 \Rightarrow k x) \land (x \geq 2 \Rightarrow F_F(u - 1)(x - 1) \lambda y.z(y + z)).$$

The resulting formula can be proved valid for $c = d = 1$. □

The following example involves a partial application of a predicate defined by $\mu$.

**Example 6** Consider the formula $x + y \leq 0 \lor F \mu x y z k = x + y \leq 0 \lor F \mu x$. Here, to approximate the formula $F \mu$, the number of unfoldings should be at least $x + y + 1$, but the value of $y$ is not available in the partial application $F \mu$. To remedy the problem, it suffices to $\eta$-expand $F \mu$ and replace the main formula with $\forall x. x \geq 0 \Rightarrow G(y,F(x,y))$. We can then apply the approximation as sketched above, and obtain $\forall x. x \geq 0 \Rightarrow G(y,F(c_1|x + c_2|y + d)(x,y))$, where:

$$G f y = \nu f y \land G f (y + 1) \land F x y = \mu x + y \leq 0 \lor F (x - 1)y.$$

The resulting formula can be proved valid for $c_1 = c_2 = d = 1$. In this manner, each partial application of a least-fixpoint predicate is $\eta$-expanded before the predicate is approximated by a greatest-fixpoint predicate. □

**Remark 2** The method above is not sufficient, for example, for proving the termination of Ackermann function:

$$\text{let rec ack m n k = if } y = 0 \text{ then } k(z+1) \text{ else if } z = 0 \text{ then } \text{ack } (y-1) 1 k \text{ else } \text{ack } y (z-1) (\lambda r.x \rightarrow \text{ack } (y-1) x k).$$

The termination of $\text{ack m n (fun r->())}$ is expressed by: $\text{ack m n } \lambda x.\text{true}$, where

$$\text{Ack } y z k = \mu_y (y = 0 \Rightarrow k(z + 1)) \land (y \neq 0 \land z = 0 \Rightarrow \text{Ack } (y - 1) 1 k) \land (y = 0 \land z = 0 \Rightarrow \text{Ack } y (z - 1) (\lambda x.\text{Ack } (y - 1) x k)).$$

Adding a single parameter $u$ to count the number of unfoldings

$$\text{Ack } u y z k = \nu_y u > 0 \land ((y = 0 \Rightarrow k(z + 1)) \land (y \neq 0 \land z = 0 \Rightarrow \text{Ack } u - 1)(y - 1) 1 k) \land (y = 0 \land z = 0 \Rightarrow \text{Ack } (u - 1)(y - 1) x k))$$

does not work, since the depth of the recursive calls of $\text{ack } y z k$ is not linear in $y$ and $z$. As suggested by Kobayashi et al. [19] for the first-order case, to deal with the example above, we need to prepare two counters and approximate $\text{Ack } m n (\lambda x.\text{true})$ by $\text{Ack } (c|m| + c|n| + d)(c|m| + c|n| + d)$ $m n (\lambda x.\text{true})$, where:

$$\text{Ack } u v w y z k = \nu_y u > 0 \land v > 0 \land ((y = 0 \Rightarrow k(z + 1)) \land (y \neq 0 \land z = 0 \Rightarrow \text{Ack } u v w u - 1)(y - 1) 1 k) \land (y \neq 0 \land z \neq 0 \Rightarrow \text{Ack } u v w u - 1)(y - 1) x k))$$

$$\text{Ack } u v w y z k = \nu_y u v w (u - 1) y z k \quad \text{ (decrement } u_1, \text{ or)}$$

$$\forall \text{ Ack } u v w (u - 1) (c|y| + c|z| + d) y z k \quad \text{ (decrement } u_1, \text{ and reset } u_2).$$
In general, given a formula $X \tilde{v}$ where $X$ is defined by: $X \tilde{y} = \mu \varphi(X)$, we can approximate it with $\forall u_{k-1}, \ldots, u_0 \geq c(u_j \Sigma|v_i|) + d. X_{RC} u_{k-1} \cdots u_0 \tilde{v}$, where:

$$X_{RC} u_{k-1} \cdots u_0 \tilde{y} = \nu \\
(u_{k-1} \geq 0 \wedge \cdots \wedge u_0 \geq 0) \land \\
(\forall \lambda \tilde{y}'. \forall u'_{k-2}, \ldots, u'_0 \geq c(\Sigma_{0 \leq j < k} u_j + \Sigma|y'_i|) + d. X_{RC} (u_{k-1} - 1) u'_{k-2} \cdots u'_0 \tilde{y}' \\
\lor \forall u'_{k-3}, \ldots, u'_0 \geq c(\Sigma_{0 \leq j < k} u_j + \Sigma|y'_i|) + d. X_{RC} u_{k-1} (u_{k-2} - 1) u'_{k-3} \cdots u'_0 \tilde{y}' \\
\lor \cdots \lor X_{RC} u_{k-1} \cdots u_1 (u_0 - 1) \tilde{y}').$$

Here, the notation $\forall u_{k-1}, \ldots, u_0 \geq c. \varphi$ abbreviates $\forall u_{k-1}, \ldots, u_1. u_{k-1} \geq e \land \cdots \land u_1 \geq e \Rightarrow \varphi$; in particular, $e$ is the lower-bound for all the variables $u_{k-1}, \ldots, u_1$. As we discuss in Section 4, the basic method with a single counter $u$ is analogous to (but strictly more powerful than) termination verification using single linear ranking functions, and the extension with multiple counters is strictly more powerful than methods based on lexicographic linear ranking functions $[4, 5]$ and disjunctive well-founded relations based on linear ranking functions. Since the extension with multiple counters is orthogonal to the extensions discussed below, we focus on the method using a single counter below. □

Example 7 Let us consider the formula:

$$(\nu f. \lambda x. (\mu g. \lambda y. (y = 0 \land f(x + 1)) \lor (y \neq 0 \land g(y - 1)))x)0,$$

whose “alternation depth” (as defined for the modal $\mu$-calculus) is 2. The formula corresponds to the property that the function $f$ is called infinitely often in the following OCaml-like program:

```ocaml
let rec f x =
  let rec g y = if y=0 then f(x+1) else g(y-1) in g x
in f 0.
```

By our approximation, we obtain $F0$, where:

$$Fx = \nu G(c|x| + d) x x \quad Guy = \nu u > 0 \land ((y = 0 \land f(x + 1)) \lor (y \neq 0 \land G(u - 1) x(y - 1))$$

(the second parameter $x$ of $G$ is introduced by lambda lifting), which can be proved valid for $c = d = 1$. □

Remark 3 From the viewpoint of computability theory, our approach of reducing $HFL(Z)$ validity checking to $\nu HFL(Z)$ validity checking has the following fundamental limitation. The $HFL(Z)$ validity checking problem is $\Pi^1_1$-hard and $\Sigma^1_1$-hard, since the fair termination problem (which is $\Pi^1_1$-complete) and its dual can be reduced to $HFL(Z)$ validity checking; see also $[2]$. In contrast, (the validity checking problem for) the $\nu HFL(Z)$-formula obtained by our reduction belongs to $\Pi^0_2$ (in other words, the set of valid $\nu HFL(Z)$ formulas is co-recursively enumerable), since the validity of a $\nu HFL(Z)$ formula can be disproved by unfolding the greatest fixpoint formulas a finite number of times and showing the resulting formula is invalid. This implies that there is no complete, effective procedure to reduce $HFL(Z)$ validity checking to $\nu HFL(Z)$ validity checking. Despite this theoretical limitation, however, as reported in Section 3 (where the benchmark set includes instances of the fair termination problem), our method can solve many instances of the $HFL(Z)$ validity checking problem that have been obtained from actual program verification problems. This kind of phenomenon has often been observed in the context of automated program verification: the fair termination verification problem (which is $\Pi^1_1$-complete) has been solved by a reduction to the safety property verification problem (which is $\Pi^0_2$-complete, hence much easier in theory) in $[2, 27]$. See also the discussion in Section 4. □
Remark 4 A reader may expect a syntactic characterization of the class of HFL(Z) formulas for which our method is complete with respect to the hypothetical completeness of the backend solver for νHFL(Z) validity checking. Our method is indeed complete for the μ-only fragment of HFL(Z) (i.e., the fragment without the greatest fixpoint operators), in that given a closed valid μ-only formula ϕ, our procedure eventually terminates and concludes that the formula is valid (see [22], Lemma 6). It seems difficult to give a clear syntactic characterization of a larger, more useful class of HFL(Z) formulas for which our method is complete. As discussed in Example 4 universal and existential quantifiers ∀x.p(x) and ∃x.p(x) can be expressed by Forall p and Exists p respectively, where:

\[
\begin{align*}
\text{Forall } p &= _\nu p \land \text{Forall } (\lambda x.p(x - 1)) \land \text{Forall } (\lambda x.p(x + 1)) \\
\text{Exists } p &= _\mu p \lor \text{Exists } (\lambda x.p(x - 1)) \lor \text{Exists } (\lambda x.p(x + 1))
\end{align*}
\]

By passing around the predicates Forall and Exists above through higher-order predicates, one can express arbitrary nesting of quantifiers to realize any \(\Sigma_n^0\) and \(\Pi_n^0\) formulas (for any n) without any syntactic nesting of greatest and least fixpoint operators. In contrast, as mentioned in Remark 5, the νHFL(Z)-formula obtained by our reduction belongs to \(\Pi_1^0\). Instead of trying to give a syntactic characterization, in Section 2 we compare the class of formulas for which our method is complete with those for which previous representative methods are complete, and show that the former is strictly larger than the latter. □

3.3 Adding Extra Arguments for Higher-Order Values

To deal with higher-order predicates, we need to extend the basic method to take function arguments into account. Let us consider the formula \(\text{All } (\lambda k.k \cdot 0)\), where All and F are defined by:

\[
\begin{align*}
\text{All } x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\nu F x \land \text{All } (\text{Succ } x) \\
\text{Succ } x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\nu x(\lambda y.k(y + 1))
\end{align*}
\]

\[
\begin{align*}
F x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\mu x(\lambda y.y = 0) \lor F (\text{Pred } x) \\
\text{Pred } x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\nu x(\lambda y.k(y - 1)).
\end{align*}
\]

This is a higher-order variant of \(\forall y. y \geq 0 \Rightarrow (\mu f^\text{int} \rightarrow \ast. \lambda y.y = 0 \lor f(y - 1))y\) considered in Example 4 where an integer y has been replaced by a higher-order-predicate \(\lambda k.k\) of type \((\text{int} \rightarrow \ast) \rightarrow \ast\).

Since F is defined by μ, we remove it by approximating it with ν. The basic translation in Section 2 would yield:

\[
\begin{align*}
\text{All}^\prime x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\nu F' d x \land \text{All}^\prime (\text{Succ } x) \\
F' u x(\text{int} \rightarrow \ast) \rightarrow \ast &= _\mu u > 0 \land (x(\lambda y.y = 0) \lor F' (u - 1) (\text{Pred } x)).
\end{align*}
\]

(We have omitted the definitions of Succ and Pred as they are unchanged.) Here, the argument u of F' represents the number of unfoldings for the original predicate F, and the bound d for u is a constant. This is because there is no integer variable in the scope of the body of All. Since the value of u should actually be greater than the value represented by x, the formula \(\text{All}^\prime (\lambda k.k \cdot 0)\) is invalid; thus, we fail to prove the validity of the original formula \(\text{All } (\lambda k.k \cdot 0)\).

To remedy the problem above, we add to \(\text{All}^\prime\) an extra integer argument that represents information about x. We thus refine the approximation of \(\text{All } (\lambda k.k \cdot 0)\) to \(\text{All}^\prime \prime (d', \lambda k.k \cdot 0)\), where \(\text{All}^\prime \prime\) is defined by:

\[
\begin{align*}
\text{All}^\prime \prime (v_x^\text{int}, x(\text{int} \rightarrow \ast) \rightarrow \ast) &= _\nu F' (cv_x + d) x \land \text{All}^\prime \prime (c'v_x + d', \text{Succ } x)
\end{align*}
\]

(We have omitted the definitions of Succ and Pred as they are unchanged.) Here, the argument u of F' represents the number of unfoldings for the original predicate F, and the bound d for u is a constant. This is because there is no integer variable in the scope of the body of All. Since the value of u should actually be greater than the value represented by x, the formula \(\text{All}^\prime \prime (\lambda k.k \cdot 0)\) is invalid; thus, we fail to prove the validity of the original formula \(\text{All } (\lambda k.k \cdot 0)\).

To remedy the problem above, we add to \(\text{All}^\prime \prime\) an extra integer argument that represents information about x. We thus refine the approximation of \(\text{All } (\lambda k.k \cdot 0)\) to \(\text{All}^\prime \prime \prime (d', \lambda k.k \cdot 0)\), where \(\text{All}^\prime \prime \prime\) is defined by:

\[
\begin{align*}
\text{All}^\prime \prime \prime (v_x^\text{int}, x(\text{int} \rightarrow \ast) \rightarrow \ast) &= _\nu F' (cv_x + d) x \land \text{All}^\prime \prime \prime (c'v_x + d', \text{Succ } x)
\end{align*}
\]

(We have omitted the definitions of Succ and Pred as they are unchanged.) Here, the argument u of F' represents the number of unfoldings for the original predicate F, and the bound d for u is a constant. This is because there is no integer variable in the scope of the body of All. Since the value of u should actually be greater than the value represented by x, the formula \(\text{All}^\prime \prime \prime (\lambda k.k \cdot 0)\) is invalid; thus, we fail to prove the validity of the original formula \(\text{All } (\lambda k.k \cdot 0)\).

To remedy the problem above, we add to \(\text{All}^\prime \prime \prime\) an extra integer argument that represents information about x. We thus refine the approximation of \(\text{All } (\lambda k.k \cdot 0)\) to \(\text{All}^\prime \prime \prime \prime (d', \lambda k.k \cdot 0)\), where \(\text{All}^\prime \prime \prime \prime\) is defined by:

\[
\begin{align*}
\text{All}^\prime \prime \prime \prime (v_x^\text{int}, x(\text{int} \rightarrow \ast) \rightarrow \ast) &= _\nu F' (cv_x + d) x \land \text{All}^\prime \prime \prime \prime (c'v_x + d', \text{Succ } x)
\end{align*}
\]
For technical convenience, we have extended the syntax of formulas with pairs (which can be removed by the standard currying transformation). The new argument $v_x$ of $\textit{All}''$ carries information about $x$, which is updated to $c'v_x + d'$ upon a recursive call of $\textit{All}'''$. The predicate $F'$ remains the same, and $c', d'$ are some positive integer constants. The formula $\textit{All}''' (d', \lambda k. k 0)$ can now be proved valid for $c = d = c' = d' = 1$.

The idea of adding extra arguments above has been inspired by Unno et al.'s method of adding extra arguments for relatively complete refinement type inference [39]. Unlike in the case of Unno et al.'s method [39], however, our method above satisfies the monotonicity property on extra arguments. Because the extra arguments are used only for computing the lower-bound of the number of unfoldings of $\mu$-formulas, the precision of the approximation monotonically increases with respect to the values of coefficients $c', d'$ (see Theorem 3.2 given later). Thus, like the values of $c, d$, we just need to monotonically increase the values of $c', d'$ to refine the precision of approximation. In contrast, in Unno et al.’s method, a rather complex procedure is required to infer appropriate extra arguments in a counterexample-guided manner.

### 3.4 Optimization Transformation

A remaining issue is how to decide where we should insert extra integer arguments. A naive way would be to add an extra integer argument to every function argument, but then too many arguments would be introduced, causing a burden for the backend validity checker for $\nu\text{HFL}(Z)$. For example, for the above example, the naive approach would yield:

$$\text{All} (v_x, x) =_{\nu} (F (cv_x + d) (c'v_x + d', x)) \land \text{All} (c'v_x + d', \text{Succ} (c'v_x + d', x))$$
$$F \ u (v_x, x) =_{\nu} (u > 0) \land (x(c'v_x + d', \lambda y. y = 0) \lor F (u - 1) (c'v_x + d', \text{Pred} (c'v_x + d', x)))$$
$$\text{Succ} (v_x, x) (v_k, k) =_{\nu} x(c'v_x + c'v_k + d', \lambda y. k(y + 1))$$
$$\text{Pred} (v_x, x) (v_k, k) =_{\nu} x(c'v_x + c'v_k + d', \lambda y. k(y - 1)).$$

The extra arguments $v_x, v_k$ for $F, \text{Succ}$, and $\text{Pred}$ would however be redundant, because they do not flow to the argument $u$ of $F$.

We introduce below types for representing where extra arguments should be inserted, and formalize the translation from $\text{HFL}(Z)$ formulas to $\nu\text{HFL}(Z)$ formulas as a type-based transformation.

We first extend types with tags. The sets of tagged argument types and tagged (predicate) types, ranged over by $\alpha$ and $\zeta$ respectively, are given by:

$$\alpha \text{ (tagged argument types) ::= Int } | \ (\zeta, t)$$
$$\zeta \text{ (tagged predicate types) ::= } * \ | \ \alpha \rightarrow \zeta \quad t \text{ (tags) ::= } T \ | \ F.$$

A tagged type $(\zeta, T)$ represents the type of a predicate argument for which an extra integer argument is required for an approximation of some $\mu$-formula. For example, recall the predicate $\text{All}$ in Section 3.3:

$$\text{All} x^{\text{Int} \rightarrow *) \rightarrow * =_{\nu} F x \land \text{All} (\text{Succ} x)$$
$$F x^{(\text{Int} \rightarrow *) \rightarrow * =_{\mu} \ast}.$$

The argument $x$ of $\text{All}$ should have type $((\text{Int} \rightarrow *) \rightarrow *)$, because information about $x$ is required to estimate how often $F$ should be unfolded, whereas information about the argument of $x$ is not.

We formalize the (optimized) transformation from $\text{HFL}(Z)$ formulas to $\nu\text{HFL}(Z)$ formulas as a type-based transformation relation $\Delta \vdash \phi: \zeta \rightsquigarrow \varphi'$ where: (i) $\Delta$, called an tagged type environment, is a finite map from variables to tagged argument types, (ii) $\varphi$ and $\varphi'$ are
the input and output of the transformation. We also use an auxiliary transformation relation $\Delta \vdash \varphi : \alpha \rightarrow \varphi'$ for the translation of an argument.

We need to introduce some notations to define the transformation relation. For a type environment $\Delta$ and a set $V$ of variables, we write $\Delta\downarrow V$ for the restriction of $\Delta$ to $V$, i.e., $\{x : \alpha \in \Delta \mid x \in V\}$. We write $\text{Tags}(\Delta)$ for the set of outermost tags in $\Delta$, defined by:

$$\text{Tags}(\Delta) = \{t \mid x : (\zeta, t) \in \Delta\}.$$  

For tagged argument types $\alpha$ and $\alpha'$, we write $\alpha \approx \alpha'$ when they are identical except their outermost tags, i.e., if either $\alpha = \alpha' = \text{Int}$, or $\alpha = (\zeta, t)$ and $\alpha = (\zeta, t')$ for some $\zeta, t$, and $t'$. For tagged types $\alpha$ and $\zeta$, we write $\text{ST}(\alpha)$ and $\text{ST}(\zeta)$ for the simple types obtained by removing the tags. We also write $\text{ST}(\Delta)$ for the simple type environment defined by $\text{ST}(x_1 : \alpha_1, \ldots, x_k : \alpha_k) = x_1 : \text{ST}(\alpha_1), \ldots, x_k : \text{ST}(\alpha_k)$. We sometimes write $\text{let } x = x_1 \text{ in } x_2$ for $(\lambda x.x_1)x_2$.

The transformation relations are defined by the rules in Fig. [3.1] The first two rules are for the translation of arguments. As specified in Tr-TagT, if the tag is T, then we add an extra argument $\text{exarg}(\Delta\downarrow_{\text{ST}(\varphi)}\varphi)$, where $\text{exarg}$ is defined by\footnote{For the sake of simplicity, we do not distinguish between the coefficients $c, d$ for estimating the number of unfoldings of $\mu$-formulas, and $c', d'$ for computing extra arguments. The actual implementation reported in Section [3] distinguishes between $c, d$ and $c', d'$.}

$$\text{exarg}(\Delta) = d + c(\sum_{x : \text{Int} \in \Delta} |x| + \sum_{x : (\zeta, T) \in \Delta} |x|).$$

It is a linear combination of (the absolute value of) original integer variables $x$ and auxiliary integer variables $v_x$. We fix the name of the auxiliary integer variable associated with $x$ to $v_x$, and assume that it does not clash with the names of other variables. We ensure that $v_x$ always takes a non-negative integer value, so that we need not take the absolute of $v_x$ in $\text{exarg}(\Delta)$. The condition $\text{Tags}(\Delta\downarrow_{\text{ST}(\varphi)}\varphi) \subseteq \{T\}$ requires that all the free variables of $\varphi$ are either integer variables or tagged with T, so that the extra argument $\text{exarg}(\Delta)$ can be properly calculated.

The rules from Tr-VAR to Tr-APPINT just transform formulas in a compositional manner, with integer expressions unchanged. In Tr-ABS, $p_{x, \alpha}$ denotes the pattern defined by:

$$p_{x, \alpha} = \begin{cases} (v_x, x) & \text{if } \alpha \text{ is of the form } (\zeta, T) \\ x & \text{otherwise} \end{cases}$$

For example, we have

$$\emptyset \vdash \lambda x.x : (\text{Int} \rightarrow \star, T) \rightarrow \star \rightarrow \lambda(v_x, x).x \ 1.$$  

Here, $\emptyset$ denotes the empty type environment. In the rule Tr-Nu, the auxiliary integer variable $v_x$ associated with $x$ is prepared when $t = \text{T}$. It is necessary in a case where $x$ is passed to another function. For example, $\nu x.f\ x$ (where $f : (\text{Int} \rightarrow \star, T) \rightarrow \text{Int} \rightarrow \star$) is translated to

$$\nu x.\text{let } v_x = cv_f + d \text{ in } f(v_x, x).$$

The key rule is Tr-Mu. To see how $\mu x.\varphi$ should be transformed, let us consider $(\mu x.\varphi)\varphi_1 \cdots \varphi_n$, where $\mu x.\varphi$ is applied to actual arguments $\varphi_1 \cdots \varphi_n$. We estimate the number of unfoldings of $\mu x.\varphi$ by gathering information from $(\mu x.\varphi)\varphi_1 \cdots \varphi_n$. Thus, all the predicate variables in $\mu x.\varphi$ and arguments should be tagged with T, as required by the third premise $\text{Tags}(\Delta, y_1 : (\alpha'_1, T), \ldots, y_n : (\alpha'_n)\downarrow_{\text{ST}(\varphi)\varphi(y_1 \cdots y_n)} \varphi(y_1 \cdots y_n) \subseteq \{T\})$. To transform the subformula $\varphi$, however, we need not require that the arguments of $x$ should be tagged with T, when they are not
passed to another least fixpoint formula in \( \varphi \). Thus, the types of arguments of \( \mu x.\varphi \) and those of \( x \) (inside \( \varphi \)) may be different in their outermost tags, as indicated in the second premise \( \alpha \approx \alpha' \). For example, it is allowed that \( \mu x.\varphi \) has type \( (\text{Int} \rightarrow \star, T) \rightarrow \star \) but \( x \) has type \( (\text{Int} \rightarrow \star, F) \rightarrow \star \) in \( \varphi \). The fourth premise \( (\varphi'' \equiv \cdots) \) is analogous to the second premise of the rule Tr-NU explained above. The last premise \( (\varphi' \equiv \cdots) \) takes care of the actual approximation of the \( \mu \)-formula by \( \nu \)-formula. The number of unfoldings of the \( \nu \)-formulas is represented by \( \text{exary}(\Delta_{\text{FV}}(\mu x.\varphi)) \), and it is passed through the extra parameter \( u \).

Finally, Tr-Sub is the rule for subsumption, which allows, for example, to convert a formula of type \( (\zeta, F) \rightarrow \star \) to that of type \( (\zeta, T) \rightarrow \star \) (but not in the opposite direction). The subtyping relation \( \zeta' <: \zeta \rightarrow \varphi \) is defined in Fig. 4. As usual, the subtyping relation on predicate types is contravariant in the argument type, and covariant in the return type. Since we need the corresponding coercion function to achieve the transformation, we have defined the subtyping relation as a ternary relation \( \zeta' <: \zeta \rightarrow \varphi \), where \( \varphi \) is a function to convert a formula of type \( \zeta' \) to that of \( \zeta \).

**Example 8** Recall the example of the formula \( \text{All} (\lambda k.0) \) in Section [6.8]. In the standard (non-equational) notation, the formula is expressed by \( \varphi_{\text{All}} (\lambda k.0) \) where:

\[
\varphi_{\text{All}} \triangleq \nu \text{All}.\lambda x.\varphi_F x \land \text{All} (\varphi_{\text{Succ}} x) \\
\varphi_{\text{Succ}} \triangleq \lambda x.\lambda y.(\lambda y.0) \lor F (\varphi_{\text{Pred}} x) \\
\varphi_{\text{Pred}} \triangleq \lambda x.(\lambda y.0) \lor F (\varphi_{\text{Pred}} x).
\]

Let \( \zeta_{\text{Succ}} \) be:

\[
( (\text{Int} \rightarrow \star, F) \rightarrow \star, F ) \rightarrow (\text{Int} \rightarrow \star, F) \rightarrow \star.
\]

Then we have:

\[
\emptyset \vdash \varphi_{\text{Succ}} : \zeta_{\text{Succ}} <: \varphi_{\text{Succ}} \quad \emptyset \vdash \varphi_{\text{Pred}} : \zeta_{\text{Succ}} <: \varphi_{\text{Pred}}.
\]

Let \( \zeta_F \) and \( \zeta'_F \) be defined by:

\[
\zeta_F \triangleq ( (\text{Int} \rightarrow \star, F) \rightarrow \star, T ) \rightarrow \star \quad \zeta'_F \triangleq ( (\text{Int} \rightarrow \star, F) \rightarrow \star, F ) \rightarrow \star.
\]

The body of \( \varphi_F \) is transformed to itself under \( F : (\zeta'_F, F) \):

\[
F : (\zeta'_F, F) \vdash \lambda x.(\lambda y.0) \lor F (\varphi_{\text{Pred}} x) : \zeta_F' \rightarrow \lambda x.(\lambda y.0) \lor F (\varphi_{\text{Pred}} x).
\]

By using Tr-Mu, we obtain \( \emptyset \vdash \varphi_{\text{F}} : \zeta_F \rightarrow \varphi_F' \), where

\[
\varphi_F' \triangleq \lambda x.(\nu F.F x.\lambda x.u > 0 \land \lambda x.(\lambda y.0) \lor F (u - 1) (\varphi_{\text{Pred}} x)) x ((\nu x\nu d)x + (\nu x\nu d) x).
\]

Let \( \zeta_{\text{All}} \) be: \( ( (\text{Int} \rightarrow \star, F) \rightarrow \star, T ) \rightarrow \star \). Then we have \( \emptyset \vdash \varphi_{\text{All}} : \zeta_{\text{All}} <: \varphi_{\text{All}}' \) where

\[
\varphi_{\text{All}}' \triangleq \nu \text{All}.\lambda x.(\nu F.F x.\lambda x.u > 0 \land \lambda x.(\lambda y.0) \lor F (u - 1) (\text{Pred} x)) x ((\nu x\nu d)x + (\nu x\nu d) x).
\]

As a result, the whole formula \( \varphi_{\text{All}} (\lambda k.0) \) is translated to \( \varphi_{\text{All}}' (\lambda k.0) \). By rewriting the resulting formula in the equational form, we get \( \text{All} (d, \lambda k.0) \), where:

\[
\begin{align*}
\text{All} (v x, x) & = (\nu F'.(\nu x\nu d)x + (\nu x\nu d) x) \land \text{All} (cv x + d, \text{Succ} x) \\
F' u x & = (\nu u > 0 \land (\lambda y.0) \lor F (u - 1) (\text{Pred} x)) \cdots .
\end{align*}
\]

By inlining \( F' \), we can further simplify the equations to:

\[
\begin{align*}
\text{All} (v x, x) & = (\nu F ((\nu x\nu d)x + (\nu x\nu d) x) \land \text{All} (cv x + d, \text{Succ} x)) \\
F u x & = (\nu u > 0 \land (\lambda y.0) \lor F (u - 1) (\text{Pred} x)) \\
\text{Succ} x k & = (\nu x(\lambda y.0) + (\nu x(\lambda y.0) + F (u - 1) (\text{Pred} x)) \\
\text{Pred} x k & = (\nu x(\lambda y.0) + (\nu x(\lambda y.0) + F (u - 1) (\text{Pred} x)) .
\end{align*}
\]

\( \square \)
Figure 3: Type-based Transformation Rules.

\[ \Delta \vdash \varphi : \zeta \leadsto \varphi' \quad \text{Tags}(\Delta \downarrow_{\text{FV}(\varphi)}) \subseteq \{ T \} \]

**Tr-TagT**

\[ \Delta \vdash \varphi : (\zeta, T) \leadsto (\text{exary}(\Delta \downarrow_{\text{FV}(\varphi)}), \varphi') \]

**Tr-TagF**

\[ \Delta \vdash \varphi : (\zeta, F) \leadsto \varphi' \]

**Tr-Var**

\[ \Delta, x : (\zeta, t) \vdash x : \zeta \leadsto x \]

**Tr-Or**

\[ \Delta \vdash \varphi_1 : * \leadsto \varphi'_1 \quad \Delta \vdash \varphi_2 : * \leadsto \varphi'_2 \]

\[ \Delta \vdash \varphi_1 \lor \varphi_2 : * \leadsto \varphi'_1 \lor \varphi'_2 \]

**Tr-And**

\[ \Delta \vdash \varphi_1 \land \varphi_2 : * \leadsto \varphi'_1 \land \varphi'_2 \]

\[ \Delta \vdash \varphi_1 : * \leadsto \varphi'_1 \]

**Tr-Ge**

\[ \text{ST}(\Delta) \vdash_{\text{ST}} e_1 \geq e_2 : * \]

**Tr-Abs**

\[ \Delta \vdash \lambda x. \varphi : \alpha \rightarrow \zeta \leadsto \lambda p_{x, \alpha}. \varphi' \]

\[ \Delta \vdash \varphi_1 : \alpha \rightarrow \zeta \leadsto \varphi'_1 \quad \Delta \vdash \varphi_2 : \alpha \rightarrow \varphi'_2 \]

**Tr-App**

\[ \Delta \vdash \varphi : \text{Int} \rightarrow \zeta \leadsto \varphi' \quad \text{ST}(\Delta) \vdash_{\text{ST}} e : \text{Int} \]

**Tr-AppInt**

\[ \Delta, x : (\zeta, t) \vdash \varphi : \zeta \leadsto \varphi'' \]

\[ \varphi'' = \left\{ \begin{array}{ll}
\text{let } v_x = \text{exary}(\Delta \downarrow_{\text{FV}(\varphi(x))}) & \text{in } \varphi'' \quad \text{if } t = T \\
\varphi'' & \text{if } t = F
\end{array} \right. \]

**Tr-Nu**

\[ \Delta \vdash \nu x. \varphi : \zeta \leadsto \nu x. \varphi' \]

\[ \Delta, x : (\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \star, t) \vdash \varphi : \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \star \leadsto \varphi'' \]

\[ \alpha_i \equiv \alpha'_i \text{ for each } i \in \{1, \ldots, n\} \]

\[ \text{Tags}(\Delta, y_1 : \alpha'_1, \ldots, y_n : \alpha'_n) \downarrow_{\text{FV}(\varphi(y_1 \ldots y_n))} \subseteq \{ T \} \]

**Tr-Mu**

\[ \varphi'' = \left\{ \begin{array}{ll}
\varphi'' & \text{if } t = T \\
\varphi'' & \text{if } t = F
\end{array} \right. \]

\[ \varphi' = (\nu x. \lambda x. \varphi_1 \ldots z_n. u > 0 \wedge (x(u-1)/x) \varphi'' z_1 \ldots z_n) \quad \text{exary}(\Delta \downarrow_{\text{FV}(\nu x. \varphi_1 \ldots y_n)}) \]

**Tr-Mu**

\[ \Delta \vdash \mu x. \varphi : \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \star \leadsto \lambda p_{y_1, \alpha'_1} \ldots p_{y_n, \alpha'_n}. \varphi'_n p_{y_1, \alpha_1} \ldots p_{y_n, \alpha_n} \]

\[ \Delta \vdash \varphi : \zeta' \leadsto \varphi' \quad \Delta \vdash \zeta' \sim \zeta \leadsto \varphi_1 \]

**Tr-Prop**

\[ \alpha \equiv \alpha' \leadsto \varphi_1 \quad \zeta' \equiv \zeta \leadsto \varphi_2 \]

**Tr-Fun**

\[ \alpha \rightarrow (\zeta, T) \leadsto (\lambda f. \lambda p_{x, \alpha}. \varphi_2(f(\varphi_1(p_{x, \alpha})))), \varphi' \]

\[ \zeta' \sim \zeta \leadsto \varphi \]

\[ (\zeta', T) \sim (\zeta, T) \leadsto \lambda (v_y, x). (v_x, \varphi x) \]

\[ \zeta' \sim \zeta \leadsto \varphi \]

\[ (\zeta', T) \sim (\zeta, F) \leadsto \lambda (v_y, x). \varphi x \]

**Tr-TagTT**

\[ (\zeta', F) \sim (\zeta, F) \leadsto \varphi \]

\[ (\zeta', F) \sim (\zeta, F) \leadsto \varphi \]

Figure 4: Subtyping Rules.
The following theorem states that our transformation provides a sound underapproximation of HFL(Z) formulas.

**Theorem 3.1 (soundness)** Suppose $\emptyset \vdash \varphi : \star \leadsto \varphi'$. If $\varphi'$ is valid, then so is $\varphi$.

The transformation relation defined in the previous section was implicitly parameterized by the constants $c$ and $d$. To make them explicit, let us write $\emptyset \vdash_{c,d} \varphi : \star \leadsto \varphi'$. The theorem below states that the precision of the approximation is monotonic with respect to $c$ and $d$. The theorem justifies our approximation refinement process in Fig. 2, which just increases the values of $c$ and $d$.

**Theorem 3.2 (monotonicity of the approximation)** Assume $0 \leq c_1 \leq c_2$ and $0 \leq d_1 \leq d_2$. Suppose $\emptyset \vdash_{c_1,d_1} \varphi : \star \leadsto \varphi'(c_1,d_1)$ and $\emptyset \vdash_{c_2,d_2} \varphi : \star \leadsto \varphi'(c_2,d_2)$ are obtained by the same derivation except the values of $c,d$. If $\varphi'(c_1,d_1)$ is valid, then so is $\varphi'(c_2,d_2)$.

Since the transformation rules are non-deterministic, we need to compare $\varphi'(c_1,d_1)$ and $\varphi'(c_2,d_2)$ obtained by the same derivation in the theorem above. Because the shapes of possible derivations do not depend on the values of $c$ and $d$, we can keep using the same derivation during the approximation refinement process, to ensure that the approximation is always refined at each iteration of the refinement loop in Fig. 2. Proofs of the theorems above are found in a longer version of this paper [36].

**Remark 5** Theorem 3.2 guarantees that the precision of the approximation monotonically increases, but does not guarantee that the approximation is precise enough for some $c$ and $d$. Indeed, there are cases where $\emptyset \vdash_{c,d} \varphi : \star \leadsto \varphi'$ and $\varphi$ is valid but $\varphi'$ is invalid for any values of $c$ and $d$: recall Remark 2.

## 4 On the Power of Our Verification Method

As discussed in Remark 3, our reduction from HFL(Z) validity checking to $\nu$HFL(Z) validity checking (hence also the overall verification method) is necessarily incomplete. In this section, we compare our method (extended as sketched in Remark 2) with previous automated methods for temporal property verification (especially termination and fair termination), and show that our method is strictly more powerful than previous methods based on (i) (lexicographic) linear ranking functions (LLRF) and those based on (ii) disjunctive well-founded relations with linear ranking functions (DWFLR). In other words, as mentioned in Remark 4 we characterize the class of HFL(Z) formulas for which our method is (relatively) complete in terms of the classes of formulas for which previous methods are complete. As mentioned already, for first-order formulas (or programs), the idea of bounding the number of unfoldings (or recursive calls) to reduce termination/liveness properties to safety properties is not new [11, 19], but the characterization of the power of such a method in terms of the popular methods using LLRF and DWFLR is new.

The comparison with DWFLR is based on the observation that any sequence that conforms to DWFLR can be embedded into a monotonically decreasing sequence over $\mathbb{N}^k$, which may be of independent interest. Below we consider only first-order formulas, as the issue of adding extra parameters (as discussed in Sections 3.3 and 3.4) is orthogonal to the discussion below.
4.1 Methods Based on Well-Founded Relations Expressed as Linear Ranking Functions

Let us consider a formula \( X \), defined by \( X \ y = \mu \varphi(X) \) (i.e., \( X = \mu^{\varphi \rightarrow \top}X, \lambda y, \varphi(x) \)), where \( X \) does not occur in \( \varphi \), and suppose that we wish to prove that \( X \ n \) holds for every integer \( n \).

Our approach was to approximate the formula \( X \ n \) with \( \forall u \geq c|n| + d.X' u n \), where

\[
X' \ u \ y = \mu u > 0 \land \varphi(X'(u - 1)).
\]

An alternative approach (suggested, e.g., in [28, 42] for fixpoint logics) is to pick a well-founded relation \( W \), and check that the relation \( W \) holds between the arguments of recursive calls. With this approach, the formula \( X \ n \) would be replaced with \( X_{W^\infty} \), where

\[
X_{W^\infty} \ y \ p \ y = \nu W(y, y_p) \land \varphi(\lambda y'.X_{W^\infty} y'y'),
\]

and \( \infty \) denotes a maximum integer with respect to the well-founded relation \( W \). Here, the extra argument \( y_p \) has been added, which represents the argument of the previous recursive call for \( X \); thus it is checked that \( W \) holds between \( y \) and \( y_p \), and \( y_p \) has been updated to \( y \) in the recursive use of \( X_{W^\infty} \) in \( \varphi(\lambda y'.X_{W^\infty} y'y') \).

In automated verification based on the latter approach, we have to fix a method to pick an appropriate well-founded relation \( W \). The simplest approach is to select a linear ranking function \( r(y) = c \times y + d_r \), let \( W(y, y_p) \) be \( 0 \leq r(y) < r(y_p) \), and infer appropriate values for \( c_r \) and \( d_r \). If \( X_{W^\infty} \) \n is valid, then the depth of recursion without violating the relation \( W(y_p, y) \equiv 0 \leq r(y) < r(y_p) \) must be at most \( r(n) + 1 \) (i.e., \( c_r n + d_r + 1 \)). Thus, \( X' (c|n| + d) n \) is also valid, for \( c = |c_r| \) and \( d = |d_r| + 1 \). Thus, whenever the method based on linear ranking functions succeeds, our method should also succeed.

Remark 6 We have defined \( W(y, y_p) \) as \( 0 \leq r(y) < r(y_p) \) above. Alternatively, we could define \( W(y, y_p) \) as \( r(y) < r(y_p) \land 0 \leq r(y_p) \) [20], so that the value of \( r(y) \) can be negative. We use the former definition for the sake of simplicity, but the latter definition can be obtained by setting \( r'(y) = \max(0, r(y) + 1) \). This change does not affect the discussions below. In Sections 4.3 and 4.4, it suffices to increase the bound on the number of unfoldings in our approach by one, and in Section 4.5, it suffices to replace \( r(y) \) in the bound on the number of unfoldings with \( |r(y)| \). See also Example [18].

Furthermore, our method is superior to the linear ranking function approach, in the following sense:

- There are formulas (or programs) for which our method succeeds but the approach of linear ranking functions would fail. Consider the following recursive function \( f \) defined by:

\[
f \ y = \text{if } y \leq 0 \text{ then } \text{() else if } y \text{ mod } 2 = 0 \text{ then } f(y + 1) \text{ else } f(y - 3).
\]

The termination of \( f(n) \) is represented by \( X \ n \), where:

\[
X \ y = \mu(y \leq 0 \Rightarrow \text{true}) \land \\
(y > 0 \Rightarrow ((y \text{ mod } 2 = 0 \Rightarrow X(y + 1)) \land (y \text{ mod } 2 \neq 0 \Rightarrow X(y - 3)))).
\]

The formula \( X \ n \) is valid for all \( n \) (indeed, \( f \ n \) terminates for all \( n \)), but since the argument of \( X \) goes up and down (e.g., \( X(6) \rightarrow X(7) \rightarrow X(4) \rightarrow X(5) \rightarrow X(2) \rightarrow \cdots \)), there exists no linear ranking function \( r(y) \) such that \( X_{W^\infty} \) \n is valid. In contrast, since the depth of required unfoldings of \( X \) (corresponding to recursive calls for \( f \)) is at most \( |n| + 1 \), our approach succeeds for any \( c \geq 1 \) and \( d \geq 1 \).
• It is easier to systematically find appropriate values of $c$ and $d$, rather than to find the coefficients $c_r$ and $d_r$ for the ranking function. Recall that our approximation of an HFL(Z) formula by a $\nu$HFL(Z) formula is monotonic on $c$ and $d$ (Theorem 5.2); thus, we just need to monotonically increase the values of $c$ and $d$, until the verification succeeds. In contrast, the precision of the ranking function approach is not monotonic on the coefficients of ranking functions. For example, consider the termination of $fyz$ where $f$ is defined by:

$$fyz = \mu \text{if } y + z \leq 0 \text{ then } () \text{ else } f(y - 2)(z + 1).$$

Then, the ranking function $r(y, z) = y + z$ serves as a termination argument, but the ranking function $r'(y, z) = y + 2z$, which has larger coefficients, does NOT serve as a termination argument. Thus, the search for appropriate ranking functions would require some heuristics.

### 4.2 Methods Based on Lexicographic Linear Ranking Functions

The approach based on linear ranking functions discussed above is often too restrictive, and a common approach for improvement is to use lexicographic linear ranking functions [9, 5]: let $r_1, \ldots, r_k$ be a sequence of linear ranking functions, and define the well-found relation $W$ by:

$$W(y, y_p) \Leftrightarrow 0 \leq r_1(y) < r_1(y_p) \lor (0 \leq r_1(y) = r_1(y_p) \land 0 \leq r_2(y) < r_2(y_p)) \lor \cdots.$$  

For example, the termination argument for the Ackermann function (given in Remark 2) can be given by $(r_1, r_2)$, where $r_1(y, z) = y$ and $r_2(y, z) = z$.

The extension discussed in Remark 2 is at least as powerful as the method based on lexicographic linear ranking functions, as discussed below. (We consider the case for $k = 2$ for the sake of simplicity; the argument generalizes to an arbitrary sequence of linear ranking functions $r_1, \ldots, r_k$.)

Let us consider a formula $X$ defined by $X \tilde{y} = \mu \phi(X)$, where $X$ does not occur in $\phi$. With the lexicographic linear ranking functions $r_1, r_2$, $X \tilde{y}$ would be approximated by $X_{WF} \tilde{y}_0 \tilde{y}$, where $\tilde{y}_0$ may be an arbitrary argument greater than $\tilde{y}$ with respect to $W$, and

$$X_{WF} \tilde{y}_0 \tilde{y} = \nu W(\tilde{y}, \tilde{y}_p) \land \phi(\lambda \tilde{y}'.X_{WF} \tilde{y}' \tilde{y}').$$

with $W(\tilde{y}, \tilde{y}_p) \Leftrightarrow 0 \leq r_1(\tilde{y}) < r_1(\tilde{y}_p) \lor (0 \leq r_1(\tilde{y}) = r_1(\tilde{y}_p) \land 0 \leq r_2(\tilde{y}) < r_2(\tilde{y}_p)).$

The extension discussed in Remark 2 (the special case of $X_{WF}$ where $k = 2$) instead approximates $X \tilde{y}$ by $\forall u \geq c(|y_1| + \cdots + |y_k|) + d.X'u u \tilde{y}$ where $\tilde{y} = y_1, \ldots, y_k$ and $X'u_1 u_0 \tilde{y} = \nu u_1 \geq 0 \land u_0 \geq 0$

$$\land \phi(\lambda \tilde{y}'.\forall u_0' \geq c(u_1 + u_0 + |y'_1| + \cdots + |y'_k|) + d.X'(u_1 - 1) u_0' \tilde{y}' \lor X'u_1 (u_0 - 1) \tilde{y}'.

Suppose $r_1(y_1, \ldots, y_k) = a_{i,1} y_1 + \cdots + a_{i,k} y_k + b_i$ for $i \in \{1, 2\}$, and let $c = \max_{i,j}(|a_{i,j}|)$ and $d = \max(|b_1|, |b_2|)$, so that $r_1(y_1, \ldots, y_k) \leq c(|y_1| + \cdots + |y_k|) + d$. The following lemma ensures that our approximation (using $X'$) is at least as good as the method based on lexicographic linear ranking functions (using $X_{WF}$).

**Lemma 4.1.** Let $X'$ and $X_{WF}$ be the formulas as given above. For any integers $\tilde{n}_p$ and $\tilde{n}$, if $m_1 \geq r_1(\tilde{n})$ and $m_2 \geq r_2(\tilde{n})$, then $[X_{WF} \tilde{n}_p \tilde{n}] \subseteq * [X'm_1 m_2 \tilde{n}].$
Proof The proof proceeds by well-founded induction on \((r_1(\bar{n}), r_2(\bar{n}))\). Suppose \([X_{\varphi} \bar{n} \bar{y}] = \top\). Then it must be the case that \(W(\bar{n}, \bar{y})\), i.e., \(0 \leq r_1(\bar{y}) < r_1(\bar{y}_p) \lor (r_1(\bar{y}) = r_1(\bar{y}_p) \land 0 \leq r_2(\bar{y}) < r_2(\bar{y}_p))\). For every \(\bar{n}'\), if \(W(\bar{n}, \bar{n}')\) does not hold, then \([X_{\varphi} \bar{n}' \bar{y}] = \bot\). Otherwise (i.e., if \(W(\bar{n}, \bar{n}')\) holds), by the induction hypothesis, we have either \([X_{\varphi} \bar{n}' \bar{y}] \subseteq \top \top [X' m_1 (m_2 - 1) \bar{n}']\) (if \(0 \leq r_1(\bar{n}') = r_1(\bar{n}) \lor 0 \leq r_2(\bar{n}') < r_2(\bar{n})\), or \([X_{\varphi} \bar{n}' \bar{y}] \subseteq \top [X'(m_1 - 1) m_2' \bar{n}']\) (if \(0 \leq r_1(\bar{n}') < r_1(\bar{n})\)) for any \(m_2' \geq r_2(\bar{n})\), which implies

\[
\begin{align*}
[X_{\varphi} \bar{n} \bar{y}] &\subseteq \top [X_{\varphi} \bar{n} \bar{y}] \subseteq \top [X' m_1 (m_2 - 1) \bar{y}'] \lor \forall u_2' \geq c(|y_1'| + \cdots + |y_\ell'|) + d.X_{\varphi} (m_1 - 1) u_2' \bar{y}'.
\end{align*}
\]

Thus, we have

\[
[X_{\varphi} \bar{n} \bar{y}] = [W(\bar{n}, \bar{n})] \land \varphi (\lambda \bar{y}'.X_{\varphi} \bar{n} \bar{y}')
\]

as required. }

The argument above implies that our method (extended with multiple counters as discussed above and in Remark 1) is at least as powerful as the method based on lexicographic linear ranking functions. Furthermore, the two points discussed at the end of Section 4.1 also apply to the comparison with lexicographic linear ranking functions. Thus, our method is strictly more powerful, and easier to automate, than the method based on lexicographic linear ranking functions.

### 4.3 Methods Based on Disjunctive Well-Founded Relations with Linear Ranking Functions

An alternative popular approach to proving termination or other liveness properties is to use disjunctive well-founded relations [15, 23]. In the context of the HFL model checking problem, the method can be recast as the following variation of \(X_{\varphi}\) above:

\[
X_{\text{DWF}} \bar{y}_p \bar{y} = \nu D(\bar{y}, \bar{y}_p) \land \varphi (\lambda \bar{y}'.X_{\text{DWF}} \bar{y}_p \bar{y}' \land X_{\text{DWF}} \bar{y}_p \bar{y}')
\]

Here, \(D\) is a finite union of well-founded relations. The main difference from \(X_{\varphi}\) is that the arguments of recursive calls are compared between any ancestors and descendants, instead of just between parents and children. In practice, a linear ranking function is often used to represent each well-founded relation composing \(D\). It is known that the method based on disjunctive well-founded relations (DWF) with linear ranking functions (called DWFLR below) is more powerful than the method based on disjunctive well-founded relations (in fact, the example discussed at the end of Section 4.1 can be handled by the former) [9], but Cook et al. [2] have empirically shown that the latter is often more efficient than the former.

There exists an example for which our method works, but DWFLR does not. Consider the predicate \(X\) defined by:

\[
X y a p = \mu y = a \lor X ((y \times a) \mod p) a p,
\]

and suppose that we wish to prove that for any positive integer \(a > 0\) and prime number \(p\), \(X (a^2) a p\) holds. Since \(a^p \equiv a \mod p\) (Fermat’s little theorem), it suffices to approximate \(X (a^2) a p\) with \(X' (p + 1) (a^2) a p\) where

\[
X' u y a p = \nu u > 0 \land ((y = a) \lor X' (a - 1) (y \times a) \mod p) a p
\]

20
in our method. However, there exists no appropriate disjunctive well-founded relation that can be expressed as a combination of linear ranking functions.

Below we show that our method is strictly more powerful than DWFLR. Suppose that \( D = R_1 \cup \cdots \cup R_s \), where \( R_i \) is an affine function (i.e., \( r_i(y_1, \ldots, y_t) \) is of the form \( c_0 + c_1y_1 + \cdots + c_2y_t \)) and \( R_v = \{ (\tilde{v}, \tilde{w}) \mid 0 \leq r_i(\tilde{v}) < r_j(\tilde{w}) \} \). We show that any sequence \( \{\tilde{v}_i\}_{0 \leq i \leq m-1} \) such that \( \forall i, j. i < j \Rightarrow (v_i, v_j) \in D \) can be mapped to a decreasing sequence over \( \mathbb{N}^k \) (where \( \mathbb{N} \) is the set of natural numbers) with respect to the lexicographic order on \( \mathbb{N}^k \).

We first prepare some definitions. Given a set \( V \subseteq \mathbb{N}^t \), we write \( V \subseteq \mathbb{N}^t \) for the set:

\[
\{ \tilde{v} \in \mathbb{N}^t \mid \forall \tilde{v} \in V. (\tilde{v}, \tilde{w}) \in D \}.
\]

Note that \( \tilde{v}_{j+1} \in V_j \) holds for any sequence \( \{\tilde{v}_i\}_{1 \leq i \leq m} \) that satisfies the above condition, where \( V_j := \{ \tilde{v}_i \mid 1 \leq i \leq j \} \). For a tuple \( \tilde{b} = (b_1, \ldots, b_k) \) with \( b_i \in \mathbb{N} \cup \{\omega\} \), we define the base set \( B_{(b_1, \ldots, b_k)} \) by:

\[
B_{(b_1, \ldots, b_k)} := \{ \{v\} \in \mathbb{N}^t \mid \forall i \in \{1, \ldots, k\}. b_i = \omega \lor 0 \leq r_i(\{v\}) = b_i \}.
\]

For \( j \in \{0, \ldots, m-1\} \), we can construct a finite set \( S_j \subseteq (\mathbb{N} \cup \{\omega\})^k \) such that \( V_j \subseteq \bigcup_{b \in S_j} B_b \) (in which case we say \( S_j \text{ covers } V_j \)). Given \( \tilde{v}_0 \in \mathbb{N}^t \), we set \( S_0 \) to

\[
\bigcup_{i \in \{1, \ldots, k\}} \{ (\omega^i-1, x, \omega^{k-i}) \mid 0 \leq x < r_i(\tilde{v}_0) \}.
\]

Suppose that \( S_{j-1} \) covers \( V_{j-1} \) (with \( j \geq 1 \)), i.e., \( V_{j-1} \subseteq \bigcup_{b \in S_{j-1}} B_b \) and \( \tilde{v}_j \in V_{j-1} \). Since \( \tilde{v}_j \in V_{j-1} \subseteq \bigcup_{b \in S_{j-1}} B_b \), we can pick \( \tilde{b} = (b'_1, \ldots, b'_k) \) such that \( \tilde{v}_j \in B_{b'_j} \) and let \( I_{\tilde{b}} = \{ i \in \{1, \ldots, k\} \mid b'_i = \omega \} \). If \( I_{\tilde{b}} = \emptyset \), then let \( S_j \) be \( S_{j-1} \setminus \{ b'_1 \} \). Otherwise, let

\[
S_j := (S_{j-1} \setminus \{ b'_1 \}) \cup \bigcup_{i \in I_{\tilde{b}}} \{ (b'_1, \ldots, b'_{i-1}, x, b'_{i+1}, \ldots, b'_k) \mid 0 \leq x < r_i(\tilde{v}_j) \}.
\]

Then \( V_j \) is covered by \( S_j \), as required. A concrete example of \( S_j \) is given in Example 9 below.

Now, let us define the measure \( \#S_j \) of \( S_j \) as \( \#_{S_j} \) (where \( \#_S = |\{ \tilde{b} \in S \mid |I_{\tilde{b}}| = i \}| \)). Intuitively, \( \#_S \) denotes the number of \( i \)-dimensional hyperplanes used to cover \( V_j \). By the construction of \( S_j \) above, \( \#_{S_0}, \#_{S_1}, \ldots \) forms a monotonically decreasing sequence with respect to the lexicographic ordering on \( \mathbb{N}^k \). Furthermore, whenever an \( i \)-dimensional hyperplane is removed, the number of \( (i-1) \)-dimensional hyperplanes added to the covering is bounded above by \( \sum_{i \in \{1, \ldots, k\}} r_i(\tilde{v}) \).

By the observation above, \( X'(r_1(\tilde{v}) + \cdots + r_k(\tilde{v}), 0^{k-1}) \tilde{v} \) is at least as good an approximation of \( X \tilde{v} \) as \( X_\text{DFR} \tilde{v} \), where \( X' \) is defined by:

\[
X'(u_{k-1}, \ldots, u_0) \tilde{y} :=_v u_{k-1} \geq 0 \land \cdots \land u_0 \geq 0 \land \varphi(\lambda \tilde{y}'.X'(u_{k-1} - 1, u_{k-2} + r_1(\tilde{y}'), \ldots, u_0) \tilde{y}')
\]

\[
\lor X'(u_{k-1}, u_{k-2} - 1, u_{k-3} + r_1(\tilde{y}'), \ldots, u_0) \tilde{y}'
\]

\[
\lor \cdots \lor X'(u_{k-1}, \ldots, u_1, u_0 - 1) \tilde{y}'.
\]

Thus, our approximation (using \( X_\text{DFR} \) in Remark 2) is at least as good as DWFLR if we set \( c \geq 1 \) and \( d \) so that \( c(|y_1| + \cdots + |y_t|) + d \geq r_1(y_1, \ldots, y_t) + \cdots + r_k(y_1, \ldots, y_t) \).
Example 9 Let \( k = \ell = 3 \), \( \tilde{v}_i (0 \leq i \leq 4) \) and \( r_j(x,y,z) \) \((j \in \{1,2,3\})\) be given as follows.

\[
\begin{align*}
\tilde{v}_0 &= (1,1,1), & \tilde{v}_1 &= (0,2,1), & \tilde{v}_2 &= (2,0,1), & \tilde{v}_3 &= (2,2,0), & \tilde{v}_4 &= (1,0,1), \\
r_1(x,y,z) &= x, & r_2(x,y,z) &= y, & r_3(x,y,z) &= z.
\end{align*}
\]

Then, \( S_j \) and \( \#S_j \) \((0 \leq i \leq 4)\) are:

\[
\begin{align*}
S_0 &= \{(0,\omega,\omega), (\omega,0,\omega), (\omega,\omega,0)\} & \#S_0 &= (3,0,0) \\
S_1 &= \{(0,1,\omega), (0,0,\omega), (0,\omega,0), (\omega,\omega,0)\} & \#S_1 &= (2,3,0) \\
S_2 &= \{(0,1,\omega), (0,0,\omega), (0,\omega,0), (1,0,\omega), (\omega,0,0), (\omega,\omega,0)\} & \#S_2 &= (1,5,0) \\
S_3 &= \{(0,1,\omega), (0,0,\omega), (0,\omega,0), (1,0,\omega), (\omega,0,0), (1,1,0), (\omega,0,1)\} & \#S_3 &= (0,7,0) \\
S_4 &= \{(0,1,\omega), (0,0,\omega), (0,\omega,0), (1,0,0), (\omega,0,0), (1,1,0), (\omega,1,0)\} & \#S_4 &= (0,6,1).
\end{align*}
\]

Example 10 Let us consider the termination of the following imperative program \[2\].

\(\text{assume}(m>0); \text{while } x>m \text{ do if } x=m \text{ then } x:=0 \text{ else } x:=x+1.\)

The termination is expressed as the validity of the HFL(Z) formula \( \forall m. \forall x. m > 0 \Rightarrow \text{Loop } x \ m \) where

\[
\text{Loop } x \ m =_\mu x = m \lor (x > m \land \text{Loop } 0 \ m) \lor (x < m \land \text{Loop } (x + 1) \ m).
\]

Cook et al. \[2\] gave the following disjunctive well-founded relation as the termination argument:

\[
(x < x_p \land 0 \leq x_p) \lor (m - x < m_p - x_p \land 0 \leq m_p - x_p).
\]

Let \( r_1(x,m) = \text{max}(0, x + 1) \) and \( r_2(x,m) = \text{max}(0, m - x + 1) \) (recall Remark \[2\]). Based on the discussion above, the formula can be approximated in our approach by: \( \forall m. \forall x. m > 0 \Rightarrow \text{Loop}' (r_1(x,m) + r_2(x,m),0) x \ m \) where:

\[
\text{Loop}' (u_1,u_0) x \ m =_\nu u_1 \geq 0 \land u_0 \geq 0 \\
\land \varphi(\lambda x'. \lambda m'. \text{Loop}' (u_1 - 1, u_0 + r_1(x', m') + r_2(x', m'))) x' \ m' \lor \text{Loop}' (u_1, u_0 - 1) x' \ m',
\]

with \( \varphi \equiv \lambda p.x = m \lor (x < m \land p \land 0 \land m) \lor (x \leq m \land p \land (x + 1) \land m) \). The part \( r_1(x,m) + r_2(x,m) \) can be replaced by \( |x| + 1 + |m - x + 1| \leq (|x| + 1) + (|m| + |x| + 1) = 2|x| + |m| + 2 \). This approximation is a conservative one obtained from the theory above; the approximation by \( \forall m. \forall x. m > 0 \Rightarrow \text{Loop}'' (|x| + |m| + 2) x \ m \) where:

\[
\text{Loop}'' u \ m =_\nu u \geq 0 \land \varphi(\text{Loop}'' (u - 1))
\]

would actually suffice. \(\square\)

## 5 Implementation and Experiments

### 5.1 Implementation

We have implemented an HFL(Z) validity checker \textsc{MuHFL} based on the method described in Section \[3\] (including the extension discussed in Remark \[2\]). We use \textsc{ReTHFL} \[17\] as the backend \(\nu\text{-HFL}(Z)\) solver\[4\]. For the sake of simplicity of the implementation, our current

\[5\]

\[6\]

\[7\]
implementation does not support the subsumption rule Tr-SUB in the type-based transformation described in Section 3.4; the lack of TR-SUB may miss some optimization opportunity in theory, but we have not observed any problem caused by it in the experiments reported in Section 5.2.

In the implementation, the coefficients for bounds for the number of unfoldings \((c, d)\), the coefficients for extra arguments \((c', d')\), and the number of counters described in Remark 2 are set as shown in Table 2 for the first four iterations of the approximation. After the fourth iteration, \(c, d, c',\) and \(d'\) are doubled for every two iterations, and the number of counters alternates between 1 and 2.

| iteration | \(c\) | \(d\) | \(c'\) | \(d'\) | the number of counters |
|-----------|------|------|------|------|------------------------|
| 1         | 1    | 2    | 1    | 1    | 1                      |
| 2         | 1    | 2    | 1    | 1    | 2                      |
| 3         | 1    | 16   | 1    | 1    | 1                      |
| 4         | 1    | 16   | 1    | 1    | 2                      |

Remark 7 As discussed already, the precision of the approximation monotonically increases with respect to \(c, d, c', d'\) and the number of counters. In practice, however, choosing large values for \(c, d, c', d'\) and the number of counters may slow down the backend \(\nu\text{HFL}(\mathbb{Z})\) solver. Thus, it would be better to choose different values for those parameters for each least fixpoint formula. Developing a better way to determine the values is left for future work. □

We additionally implemented an optimization to omit some extra arguments for consecutive higher-order arguments. For example, consider a formula \(\lambda x.\lambda y.\varphi\) whose type is \((\zeta, T) \rightarrow (\zeta, T) \rightarrow \star\). If partial applications of this formula never occur, we can transform the formula to \(\lambda(v_{x,y}, x, y).\varphi\), instead of \(\lambda(v_x, x).\lambda(v_y, y).\varphi\), because the two arguments are always passed together. We infer which extra arguments can be omitted by using a type-based analysis. In this section, we call this optimization “Optimization 2,” and the optimization described in Section 3.4 “Optimization 1.”

5.2 Evaluation

To evaluate the effectiveness of our method, we conducted the following three experiments:

- comparison with previous verification tools for temporal properties of higher-order programs [25, 24, 27, 41],
- comparison with and without the two optimizations of extra arguments, and
- further evaluation of our tool using HFL(\(\mathbb{Z}\)) formulas that are reduced from temporal property verification problems for higher-order programs [24, 13, 26, 42], which cannot be solved (at least directly) by the previous verification methods used in the first experiment. This benchmark set includes branching-time properties of higher-order programs, for which there were no automated tools to our knowledge.

Note that all the problems used in the experiments involve higher-order predicates; thus, the previous tool for the first-order fragment of HFL(\(\mathbb{Z}\)) [19] is not applicable.
Table 3: The number of solved instances per benchmark set.

| benchmark set              | no. of instances | solved by MuHFL | solved by previous tools |
|----------------------------|------------------|-----------------|--------------------------|
| termination                | 21               | 20              | 20                       |
| non-termination            | 9                | 9               | 8                        |
| fair-termination           | 10               | 10              | 8                        |
| fair-non-termination       | 16               | 16              | 15                       |
| termination-ho             | 21               | 11              | 0                        |
| fair-termination-ho        | 10               | 7               | 2                        |
| total                      | 87               | 73              | 53                       |

The experiments were conducted on a machine with Intel Xeon CPU E5-2680 v3 and 64GB of RAM. We set the timeout to 900 seconds. The benchmark instances are available at [https://github.com/hopv/hflz-benchmark](https://github.com/hopv/hflz-benchmark) and the docker image containing the source code and the binary of MuHFL is available at [https://www.kb.is.s.u-tokyo.ac.jp/vm-images/popl-2023-muhfl.tar.gz](https://www.kb.is.s.u-tokyo.ac.jp/vm-images/popl-2023-muhfl.tar.gz).

5.2.1 Comparison with previous higher-order program verification tools

We compared MuHFL with the previous automated verification tools for temporal properties of higher-order programs [25, 24, 27, 41]. We used the following six benchmark sets consisting of verification problems for OCaml programs.

- **termination**: termination verification problems taken from [25].
- **non-termination**: non-termination verification problems taken from [24].
- **fair-termination**: verification problems for fair termination, taken from [27].
- **fair-non-termination**: verification problems for fair non-termination, taken from [41].
- **termination-ho**: a variation of **termination**, where integer values have been converted to closures in a manner similar to the example in Section 3.3.
- **fair-termination-ho**: a variation of **fair-termination**, where integer values have been converted to closures.

From the benchmark sets, we excluded out instances that can be directly reduced to $\nu$HFL($Z$) formulas (i.e., formulas without the least fixpoint operator $\mu$).

For each verification problem instance in the benchmark sets, we (automatically) converted it to the HFL($Z$) validity checking problem by using the reductions in [21, 42], and ran our tool MuHFL. We compared its performance with the result of running the corresponding previous verification tool (e.g. Kuwahara et al.’s tool [25] for **termination** and **termination-ho**).

The result is summarized in Fig. 5. In the figure, “Fail” means that the tool was aborted with some error. The number of solved instances per benchmark set is shown in Table 3. In total, our tool MuHFL could solve more problems than (the combination of) the four previous tools. In particular, for the benchmark sets with more higher-order values (**termination-ho** and **fair-termination-ho**), the previous tools could solve only two instances while MuHFL solved 18 instances. We believe that the failure of MuHFL to solve the 14 instances is mainly due to the current limitations of the backend solver ReTHFL, rather than a fundamental
limitation of our approach. In fact, we confirmed that most of the 14 instances could be solved by the backend solver after some manual transformation of the \( \nu \)HFL(Z) formulas generated by our reduction from HFL(Z) to \( \nu \)HFL(Z).

As for the instances for which both the previous tools and MuHFL succeed, MuHFL were often faster than the previous tools. There are three outliers in Fig. 5 for which our tool is significantly slower: one non-termination problem and two termination problems. One of those instances requires a large value for the constant \( d \), hence requiring the number of iterations of the approximation (cf. Table 2). For the other two, the generated \( \nu \)HFL(Z) formulas belong to a class of formulas which the current backend solver ReTHFL is not good at (specifically, the class of formulas that contain disjunction on fixpoint formulas). This problem can be remedied by a further improvement of the backend solver.

Overall, our tool MuHFL outperformed the previous tools, which is remarkable, considering that the previous tools were specialized for particular verification problems (such as termination and non-termination), whereas our tool can deal with all of those verification problems in a uniform manner. More details of the experimental results are given in a longer version [36].

5.2.2 Comparison with and without the optimizations of extra arguments

To evaluate the effectiveness of the optimizations on extra arguments (Optimization 1 discussed in Section 3.3 and Optimization 2 explained in Section 5.1), we compared the running times of our tool with and without the optimizations. From the previous experiments, we picked instances for which extra arguments are required and MuHFL successfully terminated.

The comparison of the total running times (measured in seconds) is shown in Fig. 6. Fig. 7 shows the times taken by the backend solver, and Fig. 8 compares the numbers of extra parameters. The instances from murase-closure-ho to koskinen-4-ho are from fair-termination-ho, and the other instances are from termination-ho. As observed in Fig. 6 the optimizations were generally effective; the two optimizations reduced the total
running times except for sum-ho, with a maximum reduction of 2.1 seconds for binomial-ho. For all instances, the number of extra parameters was reduced to less than half by the optimizations, which we think is one of the reasons for the reduction of the running time. The optimizations were particularly effective for binomial-ho. The reason could be that the number of extra parameters for it was significantly reduced by the optimizations. The number of iterations of the approximation refinement cycle required to solve the instances was one for all instances and was not changed by the optimizations. The optimizations (in particular, Optimization 2 alone for fibonacci-ho) sometimes increased the running times by up to a factor of 12. That seems to be due to some unexpected behavior of the backend solver.

\footnote{For example, sometimes just a renaming of predicates significantly changes the running time of a backend CHC solver used inside ReTHFL.}
To test our tool for other temporal verification problems, we have collected a new benchmark set from previous papers on temporal verification of higher-order programs [13, 23, 26, 42]. We automatically converted the problems to the HFL(Z) validity checking problem by using the reduction of Watanabe et al. [42], and ran MuHFL. The results are shown in Table 4, where times are shown in seconds. The instance hofmann-2-direct comes from the work of Hofmann and Chen [13]. Specifically, it is the second example from Appendix A of the extended technical report [14]. The instances koskinen-1-direct, koskinen-2-direct, koskinen-3-direct, and koskinen-4-direct are from Figure 10 of the paper by Koskinen and Terauchi [23], where koskinen-1-direct is REDUCE, koskinen-2-direct is RUMBLE, koskinen-3-direct are EVENTUALLY GLOBAL, and koskinen-4-direct is ALTERNATE INEVITABILITY. The instances whose names contain buggy are variations of the Koskinen and Terauchi’s instances, where the original instances are modified so that the specified properties are violated. The instance lester-direct is from Appendix H.1 of the paper by Lester et al. [26]. The previous verification tools used in the first experiment cannot directly solve those instances. The last three instances, i.e., sas19-tab1-24-ho, repeat, and repeat2, are about branching-time properties of higher-order programs, for which there were no previous automated tools to our knowledge. The instance sas19-tab1-24-ho is a higher-order version of the corresponding instance used in [19] (#24 in Table 1). The instance repeat has been taken from [42] (Example 3.3) and repeat2 is a variation of it. Watanabe et al. [12] proved the validity of repeat manually by using Coq, but our tool can now prove it fully automatically.

As shown in Table 4, all the instances were successfully solved by our tool. Note that, among the previous work [13, 23, 26, 42] which those instances come from, only Lester et al. [26] implemented an actual automated verification tool, which can solve lester-direct (but not others) in 0.035 second. For the other instances, we are not aware of other fully automated tools that can directly solve them.

Figure 8: Comparison of the number of extra parameters with and without the optimizations.

5.2.3 Solving other temporal verification problems for higher-order programs
Table 4: Results and running times for solving HFL(Z) formulas translated from verification problems for higher-order programs.

| instance                        | result | running time |
|---------------------------------|--------|--------------|
| hofmann-2-direct                 | valid  | 0.27         |
| koskinen-1-buggy-direct          | invalid| 0.31         |
| koskinen-1-direct                | valid  | 0.19         |
| koskinen-2-buggy-direct          | invalid| 1.38         |
| koskinen-2-direct                | valid  | 0.70         |
| koskinen-3-buggy-1-direct        | invalid| 0.32         |
| koskinen-3-buggy-2-direct        | invalid| 0.39         |
| koskinen-3-direct                | valid  | 0.37         |
| koskinen-4-direct                | valid  | 0.38         |
| lester-direct                    | valid  | 0.22         |
| sas19-tab1-24-bo                 | valid  | 0.21         |
| repeat                          | valid  | 6.99         |
| repeat2                         | valid  | 8.05         |

6 Related Work

As already mentioned, the framework of program verification by reduction to HFL(Z) validity checking has been advocated by Kobayashi et al. [22, 42]. It can be considered a generalization of the CHC-based program verification framework [2], where higher-order predicates and fixpoint alternations are allowed in the target logic. Burn et al. [6] also considered a higher-order extension of CHC and its application to program verification, but their logic corresponds to the $\nu$HFL(Z) fragment, which does not support fixpoint alternations. Higher-order predicates are useful for modeling higher-order programs, and fixpoint alternations are useful for dealing with temporal properties. The effectiveness of the framework has been partially demonstrated for the first-order fragment of HFL(Z) [19]. There have been implementations of automated validity checkers for $\nu$HFL(Z) (a fragment of HFL(Z) without least fixpoint operators) [15, 17], and a satisfiability checker for HoCHC [9]. To our knowledge, however, there have been no tools for full HFL(Z).

Watanabe et al. ([42], Section 4.2) sketched (but have not implemented) another method for approximating least fixpoint formulas with greatest fixpoint formulas. Their method relies on the discovery of a well-founded relation on the arguments of fixpoint predicates, which is hard to automate, especially in the presence of higher-order arguments. In contrast, our approach is much easier to automate; to refine the approximation, we just need to monotonically increase constant parameters ($c, d, c', d'$ in Section 3). The idea of our approach has been inspired by the work of Kobayashi et al. [19] on the first-order fragment of HFL(Z). That idea can further be traced back to the method of Fedynkovich et al. [11] for termination analysis.

The idea of adding extra integer parameters for higher-order arguments has been inspired by Unno et al.’s work [39] on a relatively complete refinement type system, but the details on the way extra parameters are different. In particular, our method of adding extra parameters is easier to automate. We have also proposed a type-based optimization to avoid redundant extra parameters. Our type-based optimization may be considered an instance of type-based flow analysis [31, 29].

Various techniques have been proposed and implemented for automated verification of
various linear-time temporal properties of higher-order programs, including safety properties 34 37 20 35 30 32 43, termination 25, non-termination 24, fair termination 27, and fair non-termination 41. In contrast to those studies, which developed separate techniques and tools for proving different properties, our HFL(Z) validity checker serves as a common backend for all of those properties, and can also be used for the verification of branching-time properties of higher-order programs.

7 Conclusion

We have proposed an automated method for HFL(Z) validity checking, which provides a streamlined approach to fully automated verification of temporal properties of higher-order programs, and proved the soundness of our method. We have also compared our approach with previous verification methods for proving termination and liveness properties, such as those using lexicographic linear ranking functions and disjunctively well-founded relations. We have implemented a tool based on the proposed method, and confirmed its effectiveness through experiments. To our knowledge, our tool is the first automated HFL(Z) validity checker, which serves as a common backend tool for automated verification of temporal properties of functional programs.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP20H05703.

References

[1] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In N. Sharygina and H. Veith, editors, Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of Lecture Notes in Computer Science, pages 869–882. Springer, 2013.

[2] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. 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, volume 9300 of LNCS, pages 24–51. Springer, 2015.

[3] N. S. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In F. Logozzo and M. Fähndrich, editors, Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, volume 7935 of Lecture Notes in Computer Science, pages 105–125. Springer, 2013.

[4] J. C. Bradfield and I. Walukiewicz. The mu-calculus and model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, Handbook of Model Checking, pages 871–919. Springer, 2018.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In K. Etessami and S. K. Rajamani, editors, Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings, volume 3576 of Lecture Notes in Computer Science, pages 491–504. Springer, 2005.
[6] T. C. Burn, C. L. Ong, and S. J. Ramsay. Higher-order constrained Horn clauses for verification. *Proc. ACM Program. Lang.*, 2(POPL):11:1–11:28, 2018.

[7] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*, Nice, France, January 17-19, 2007, pages 265–276. ACM, 2007.

[8] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *Proceedings of PLDI 2013*, pages 219–230. ACM Press, 2013.

[9] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.

[10] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. Softw. Tools Technol. Transf.*, 3(3):250–270, 2001.

[11] G. Fedyukovich, Y. Zhang, and A. Gupta. Syntax-guided termination analysis. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *LNCS*, pages 124–143. Springer, 2018.

[12] D. Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM*, 33(1):224–248, 1986.

[13] M. Hofmann and W. Chen. Abstract interpretation from Büchi automata. In T. A. Henzinger and D. Miller, editors, *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*, pages 51:1–51:10. ACM, 2014.

[14] M. Hofmann and W. Chen. Büchi types for infinite traces and liveness. *CoRR*, abs/1401.5107, 2014.

[15] N. Iwayama, N. Kobayashi, R. Suzuki, and T. Tsukada. Predicate abstraction and CEGAR for vhfl validity checking. In D. Pichardie and M. Sighireanu, editors, *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, volume 12389 of *Lecture Notes in Computer Science*, pages 134–155. Springer, 2020.

[16] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP method for compositional and intermittent predicate abstraction. In *Proceedings of VMCAI 2006*, volume 3855 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2006.

[17] H. Katsura, N. Iwayama, N. Kobayashi, and T. Tsukada. A new refinement type system for automated vhfl validity checking. In B. C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 86–104. Springer, 2020.
[18] N. Kobayashi, É. Lozes, and F. Bruse. On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In G. Castagna and A. D. Gordon, editors, Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 246–259. ACM, 2017.

[19] N. Kobayashi, T. Nishikawa, A. Igarashi, and H. Unno. Temporal verification of programs via first-order fixpoint logic. In B. E. Chang, editor, Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings, volume 11822 of Lecture Notes in Computer Science, pages 413–436. Springer, 2019.

[20] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In M. W. Hall and D. A. Padua, editors, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 222–233. ACM, 2011.

[21] N. Kobayashi, T. Tsukada, and K. Watanabe. Higher-order program verification via HFL model checking. CoRR, abs/1710.08614, 2017.

[22] N. Kobayashi, T. Tsukada, and K. Watanabe. Higher-order program verification via HFL model checking. In A. Ahmed, editor, 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, volume 10801 of Lecture Notes in Computer Science, pages 711–738. Springer, 2018.

[23] E. Koskinen and T. Terauchi. Local temporal reasoning. In T. A. Henzinger and D. Miller, editors, 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, pages 59:1–59:10. ACM, 2014.

[24] T. Kuwahara, R. Sato, H. Unno, and N. Kobayashi. Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In D. Kroening and C. S. Pasareanu, editors, Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II, volume 9207 of Lecture Notes in Computer Science, pages 287–303. Springer, 2015.

[25] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In Z. Shao, editor, Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8410 of Lecture Notes in Computer Science, pages 392–411. Springer, 2014.

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

[27] A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. Temporal verification of higher-order functional programs. In R. Bodik and R. Majumdar, editors, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 57–68. ACM, 2016.
[28] Y. Nanjo, H. Unno, E. Koskinen, and T. Terauchi. A fixpoint logic and dependent effects for temporal property verification. In A. Dawar and E. Grädel, editors, Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018, pages 759–768. ACM, 2018.

[29] F. Nielson, H. R. Nielson, and C. Hankin. Principles of Program Analysis. Springer, 1999.

[30] C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In T. Ball and M. Sagiv, editors, Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 587–598. ACM, 2011.

[31] J. Palsberg. Type-based analysis and applications. In J. Field and G. Snelting, editors, Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE’01, Snowbird, Utah, USA, June 18-19, 2001, pages 20–27. ACM, 2001.

[32] Z. Pavlinovic, Y. Su, and T. Wies. Data flow refinement type inference. Proc. ACM Program. Lang., 5(POPL):1–31, 2021.

[33] A. Podelski and A. Rybalchenko. Transition invariants. In 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings, pages 32–41. IEEE Computer Society, 2004.

[34] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasingham, editors, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pages 159–169. ACM, 2008.

[35] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In E. Albert and S. Mu, editors, Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013, pages 53–62. ACM, 2013.

[36] K. Tanahashi, N. Kobayashi, and R. Sato. Automatic HFL(Z) validity checking for program verification. CoRR, abs/2203.07601, 2022.

[37] T. Terauchi. Dependent types from counterexamples. In M. V. Hermenegildo and J. Palsberg, editors, Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, pages 119–130. ACM, 2010.

[38] T. Tsukada. On computability of logical approaches to branching-time property verification of programs. In H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, editors, LICS ’20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020, pages 886–899. ACM, 2020.

[39] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In R. Giacobazzi and R. Cousot, editors, The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013, pages 75–86. ACM, 2013.
[40] M. Viswanathan and R. Viswanathan. A higher order modal fixed point logic. In P. Gardner and N. Yoshida, editors, CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings, volume 3170 of Lecture Notes in Computer Science, pages 512–528. Springer, 2004.

[41] K. Watanabe, R. Sato, T. Tsukada, and N. Kobayashi. Automatically disproving fair termination of higher-order functional programs. In J. Garrigue, G. Keller, and E. Sumii, editors, Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016, pages 243–255. ACM, 2016.

[42] K. Watanabe, T. Tsukada, H. Oshikawa, and N. Kobayashi. Reduction from branching-time property verification of higher-order programs to HFL validity checking. In M. V. Hermenegildo and A. Igarashi, editors, Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019, pages 22–34. ACM, 2019.

[43] H. Zhu, A. V. Nori, and S. Jagannathan. Learning refinement types. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pages 400–411. ACM, 2015.
Appendix

A  Correctness of the Transformation

We show correctness of the transformation given in Section 3.4.

We first show that the output of the transformation is a well-typed formula (which implies, in particular, extra variables are appropriately passed around). Since we have extended the syntax of the target language with pairs, we extend simple types by:

\[ \kappa \text{ (extended simple types)} ::= \text{Int} | \tau | \text{Int} \times \tau \]

and extend the typing rules in Fig. 1 with the following rules.

\[
\begin{align*}
\frac{}{\Gamma \vdash \text{ST} \; c : \text{Int}} \quad \frac{}{\Gamma \vdash \text{ST} \; \phi : \tau} \\
\frac{}{\Gamma, v_x : \text{Int}, x : \tau_1 \vdash \text{ST} \; \phi : \tau_2} \quad (\text{T-PAIR}) \\
\frac{\Gamma \vdash \text{ST} \; \lambda (v_x, x) \cdot \phi : \text{Int} \times \tau_1 \rightarrow \tau_2}{\Gamma, \nu_x \vdash \text{ST} \; (\nu_x, x) : \text{Int} \times \tau_1} \quad (\text{T-PAbs})
\end{align*}
\]

For tagged types \( \alpha \) and \( \zeta \), the corresponding simple types \( \alpha^\dagger \) and \( \zeta^\dagger \) are defined by:

\[
\begin{align*}
(\zeta, \text{T})^\dagger &= \text{Int} \times \zeta^\dagger \\
(\zeta, \text{F})^\dagger &= \zeta^\dagger \\
\text{Int}^\dagger &= \text{Int} \\
(\alpha \rightarrow \zeta)^\dagger &= \alpha^\dagger \rightarrow \zeta^\dagger.
\end{align*}
\]

We extend the operation to type environments by:

\[
(x_1 : \alpha_1, \ldots, x_k : \alpha_k)^\dagger = p_{x_1, \alpha_1} \cdot \alpha_1^\dagger \ldots p_{x_k, \alpha_k} \cdot \alpha_k^\dagger.
\]

Here, \((v_x, x) : \text{Int} \times \tau\) is considered a shorthand for \(v_x : \text{Int}, x : \tau\). For example,

\[
(x : (\ast, \text{F}), y : (\text{Int} \rightarrow \ast, \text{T}))^\dagger = x : \ast, v_y : \text{Int}, y : \text{Int} \rightarrow \ast.
\]

The following lemma states that the output of the transformation is a well-typed formula.

**Lemma A.1** If \( \Delta \vdash \phi : \tau \leadsto \phi' \) then \( \Delta^\dagger \vdash \text{ST} \; \phi' : \tau^\dagger \). In particular, \( \emptyset \vdash \phi : \ast \leadsto \phi' \) implies \( \emptyset \vdash \text{ST} \; \phi' : \ast \).

**Proof** This follows by straightforward induction on the derivations of \( \Delta \vdash \phi : \tau \leadsto \phi' \) and \( \Delta \vdash \phi : \alpha \leadsto \phi' \). \(\square\)

To prove Theorem 3.1 we extend the semantics of HFL(\( \mathbb{Z} \)) formulas defined in Section 2 with pairs introduced in Section 3.3.

\[
\begin{align*}
\Gamma \vdash \text{ST} \; \phi : \text{Int} \times \tau & = \{(n, w) \mid n \in \mathbb{Z}, w \in \{[\tau]\}\} \\
\Gamma \vdash \text{ST} \; (c, \phi) : \text{Int} \times \tau & = \{((n, w), (n, w')) \mid (\{\text{Int} \times \tau\}) \times (\{\text{Int} \times \tau\}) \} \\
\Gamma \vdash \text{ST} \; \lambda (v_x, x) \cdot \phi : \text{Int} \times \tau_1 \rightarrow \tau & = \lambda (n, w) \in \mathbb{Z} \times \{\tau_1\} \cdot \Gamma, v_x : \text{Int}, x : \tau_1 \vdash \text{ST} \; \phi : \tau \{\rho \{v_x \mapsto n, x \mapsto w\}\}
\end{align*}
\]
We define the approximation relation \( \succeq_\zeta \subseteq (\mathsf{ST}(\zeta)) \times (\{\zeta\}) \) by:
\[
\begin{align*}
\succeq_{\mathsf{int}} &= \{(n, n) \mid n \in \mathbb{Z}\} \\
\succeq_\tau &= \{\{\top, \top\}, \{\top, \top\}, \{\bot, \bot\}\} \\
\succeq_{(\zeta, F)} &= \succeq_\zeta \\
\succeq_{(\zeta, T)} &= \{(w, (n, w')) \mid n \in \mathbb{Z}, w \succeq w'\} \\
\succeq_{\alpha \rightarrow \zeta'} &= \{(f, f') \mid \forall w, w', w \succeq \alpha w' \Rightarrow f w \succeq f' w'\}.
\end{align*}
\]
For \( \Delta \), we define \( \succeq_\Delta \subseteq (\mathsf{ST}(\Delta)) \times (\{\Delta\}) \) by:
\[
\rho \succeq_\Delta \rho' \iff \forall x \in \mathsf{dom}(\Delta), \rho(x) \succeq_\zeta \rho'(p_x, \Delta(x)).
\]

**Lemma A.2** If \( \Delta \vdash \varphi : \zeta \rightarrow \varphi' \) and \( \rho \succeq_\Delta \rho' \), then
\[
[\mathsf{ST}(\Delta) \vdash_{\mathsf{ST}} \varphi : \mathsf{ST}(\zeta)] \rho \succeq_\zeta [\Delta \vdash_{\mathsf{ST}} \varphi' : \zeta'] \rho'.
\]

**Proof** This follows by induction on the derivation of \( \Delta \vdash \varphi : \zeta \rightarrow \varphi' \), with case analysis on the last rule. Since the other cases are trivial, we discuss only the case for Tr-Mu. Suppose that the last rule used for deriving \( \Delta \vdash \varphi : \zeta \rightarrow \varphi' \) is Tr-Mu. Then, we have:
\[
\begin{align*}
\varphi &= \mu x. \varphi_1 \\
\zeta &= \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow * \\
\zeta' &= \alpha_1' \rightarrow \cdots \rightarrow \alpha_n' \rightarrow * \\
\Delta, x : (\zeta', t) \vdash \varphi_1 : \zeta' \rightarrow \varphi_1'' \\
\alpha_i &\approx \alpha_i' \text{ for each } i \in \{1, \ldots, n\} \\
\text{Tags}(\Delta, y_1 : \alpha_1, \ldots, y_n : \alpha_n) \downarrow_{\mathsf{FV}(\varphi_1 y_1 \cdots y_n)} \subseteq \{T\}
\end{align*}
\]

\( \varphi'' = \begin{cases} 
\text{let } v_x = \mathsf{exarg}(\Delta \downarrow_{\mathsf{FV}(\mu x. \varphi_1)}) \text{ in } \varphi''_1 
& \text{if } t = T \\
\varphi''_1 
& \text{if } t = F
\end{cases} \)
\( \varphi_1' = (\mu x. \lambda z_1 \cdots z_n. u > 0 \land (x(u - 1)/z)_1 \cdots z_n) \mathsf{exarg}(\Delta \downarrow_{\mathsf{FV}(\mu x. \varphi_1)}) \)
\( \varphi_1'' = \lambda p_{y_1, \alpha_1} \cdots p_{y_n, \alpha_n}. \varphi_1' p_{y_1, \alpha_1} \cdots p_{y_n, \alpha_n} \)

By the induction hypothesis, for any \( w, w' \) such that \( w \succeq_{(\zeta', t)} w' \), we have:
\[
[\mathsf{ST}(\Delta), x : \mathsf{ST}(\zeta') \vdash_{\mathsf{ST}} \varphi_1 : \mathsf{ST}(\zeta')] \rho(x \mapsto w) \succeq_\zeta [\Delta \vdash_{\mathsf{ST}} \varphi_1'' : \zeta''] \rho'(x \mapsto w').
\]

Let \( w'' \) be the second element of \( w' \) if \( t = T \), and \( w'' = w' \) otherwise. Then, from the relation above and the definition of \( \varphi''_1 \), we obtain:
\[
[\mathsf{ST}(\Delta), x : \mathsf{ST}(\zeta') \vdash_{\mathsf{ST}} \varphi_1 : \mathsf{ST}(\zeta')] \rho(x \mapsto w) \succeq_\zeta [\Delta \vdash_{\mathsf{ST}} \varphi''_1 : \zeta''] \rho'(x \mapsto w'').
\]

Therefore, we have:
\[
\lambda w \in ([\mathsf{ST}(\zeta')] \downarrow \cdot [\mathsf{ST}(\Delta), x : \mathsf{ST}(\zeta') \vdash_{\mathsf{ST}} \varphi_1 : \mathsf{ST}(\zeta')] \rho(x \mapsto w) \\
\succeq_{(\zeta', F) \rightarrow \zeta'} \lambda w'' \in ([\zeta''] \downarrow \cdot [\Delta \vdash_{\mathsf{ST}} \varphi''_1 : \zeta''] \rho'(x \mapsto w')).
\]

Let \( m \) be \( [\Delta \vdash_{\mathsf{ST}} \mathsf{exarg}(\Delta \downarrow_{\mathsf{FV}(\mu x. \varphi_1 y_1 \cdots y_n)}) : \mathsf{Int}] \rho' \). It follows by easy induction on \( m \) that
\[
[\Delta \vdash_{\mathsf{ST}} \varphi_1'' : \zeta''] \rho' = (\lambda w \in ([\zeta''] \downarrow \cdot [\Delta \vdash_{\mathsf{ST}} \varphi_1'' : \zeta''] \rho'(x \mapsto w))''(\zeta'').
\]
Thus, we have:

\[ \text{ST}(\Delta) \vdash_{\text{ST}} \varphi : \text{ST}(\zeta) \quad \rho \]
\[ \equiv_{\text{ST}(\zeta)} (\lambda w \in (\text{ST}(\zeta')) \cdot \text{ST}(\Delta), x : \text{ST}(\zeta') \vdash_{\text{ST}} \varphi_1 : \text{ST}(\zeta') \quad \rho\{x \mapsto w\})^m(\downarrow_{\text{ST}(\zeta)}) \]
\[ \leq_{\zeta'} (\lambda w \in (\downarrow_{\zeta'}), [\Delta^1, x : \zeta' \vdash_{\text{ST}} \varphi'_1 : \zeta' \quad \rho'\{x \mapsto w\})^m(\downarrow_{\zeta'}) \]
\[ = [\Delta^1 \vdash_{\text{ST}} \varphi'_1 : \zeta' \quad \rho'] \]

Therefore, we have \[ [\text{ST}(\Delta) \vdash_{\text{ST}} \varphi : \text{ST}(\zeta)] \quad \rho \leq_{\zeta} [\Delta^1 \vdash_{\text{ST}} \varphi' : \zeta'] \quad \rho' \]
as required. \( \square \)

Theorem 3.1 follows as an immediate corollary of the above lemma.

Proof [Proof of Theorem 3.1] A special case of Lemma A.2 where \( \Delta = \emptyset \) and \( \zeta = * \). \( \square \)

To prove Theorem 3.1, we define another family of relations \( \{\leq_{\zeta}\}_\zeta \) parameterized by \( \zeta \).

\[ \leq_{\text{Int}} = \{(n, n) \mid n \in \mathbb{Z} \} \]
\[ \leq_{\zeta} = \{(\bot, \bot), (\bot, \bot), (\bot, \bot)\} \]
\[ \leq_{(\zeta, \emptyset)} = \leq_{\zeta} \]
\[ \leq_{(\zeta, T)} = \{(n, n), (n, n') \mid n \leq n', w \leq_{\zeta} w'\} \]
\[ \leq_{\alpha, \zeta} = \{(f, f'), \forall u, w, \alpha \leq_u w' \Rightarrow f w \leq_{\zeta} f' w'\} \]

We write \( \rho \leq_{\Delta} \rho' \) if \( \rho(p_x, \Delta(x)) \leq_{\Delta(x)} \rho'(p_x, \Delta(x)) \) for every \( x \in \text{dom}(\Delta) \).

Lemma A.3 Suppose \( \Delta \vdash_{c_1, d_1} \varphi : \zeta \rightarrow \varphi'(c_1, d_1) \) and \( \Delta \vdash_{c_2, d_2} \varphi : \zeta \rightarrow \varphi'(c_2, d_2) \) are derived from the same derivation except the values of \( c, d \). Suppose also \( \rho \leq_{\Delta} \rho' \). If \( 0 \leq c_1 \leq c_2 \) and \( 0 \leq d_1 \leq d_2 \), then

\[ [\Delta^1 \vdash_{\text{ST}} \varphi'(c_1, d_1) : \zeta'] \quad \rho \leq_{\zeta} [\Delta^1 \vdash_{\text{ST}} \varphi'(c_2, d_2) : \zeta'] \quad \rho' \]

Proof This follows by induction on the derivation of \( \Delta \vdash_{c_1, d_1} \varphi : \zeta \rightarrow \varphi'(c_1, d_1) \), with case analysis on the last rule. We discuss only the case for Tr-Mu, since the other cases are trivial. In the case for Tr-Mu, we have:

\[ \varphi = \mu x.\varphi_1 \]
\[ \zeta = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow * \]
\[ \zeta' = \alpha'_1 \rightarrow \cdots \rightarrow \alpha'_n \rightarrow * \]
\[ \Delta, x : (\zeta', t) \vdash_{c,d} \varphi_1 : \zeta' \rightarrow \varphi_1^{n(c,d)} \]
\[ \alpha_i \approx \alpha'_i \text{ for each } i \in \{1, \ldots, n\} \]

Tags((\Delta, y_1 : \alpha_1, \ldots, y_n : \alpha_n) \downarrow_{\text{TV}}^{c_1, d_1} \varphi_1) = \{T\}

\[ \varphi_1^{n(c,d)} = \begin{cases} \text{let } v_x = \text{exarg}^{c,d}(\Delta) & \text{in } \varphi_1^{n(c,d)} \quad \text{if } t = T \\ \varphi_1^{n(c,d)} & \text{if } t = F \end{cases} \]

\[ \varphi_1^{n(c,d)} = \begin{cases} (\nu x.\lambda u.\lambda z_1 \cdots z_n, u > 0 \wedge & (n x - 1)/x) \varphi_1^{n(c,d)} z_1 \cdots z_n \text{ exarg}^{c,d}(\Delta) & \text{if } t = T \\ \lambda p_{y_1, \alpha_1} \cdots p_{y_n, \alpha_n} \varphi_1^{n(c,d)} & \text{if } t = F \end{cases} \]

for \( (c, d) \in \{(c_1, d_1), (c_2, d_2)\} \). Here, we have made \( c, d \) explicit in \text{exarg}. Suppose \( \rho \leq_{\Delta} \rho' \).

By the induction hypothesis, for any \( w \leq_{\zeta', t} w' \), we have

\[ [\Delta^1, p_x, (\zeta', t) \vdash_{\text{ST}} \varphi_1^{n(c_1, d_1)} : \zeta'] (\rho(p_x, (\zeta', t) \Rightarrow w)) \]
\[ \leq_{\zeta'} [\Delta^1, p_x, (\zeta', t) \vdash_{\text{ST}} \varphi_1^{n(c_2, d_2)} : \zeta'] (\rho'(p_x, (\zeta', t) \Rightarrow w')). \]
Let \( w_1, w'_1 \) be the second component of \( w, w' \) if \( t = T \) and \( w_1 = w, w'_1 = w' \) otherwise. Since

\[
[(\Delta) \vdash_{\text{st}} \text{exarg}^{(c_1, d_1)}(\Delta_{\text{FV}(\mu x. \varphi_1)}) : \text{Int}] \rho
\leq [(\Delta) \vdash_{\text{st}} \text{exarg}^{(c_2, d_2)}(\Delta_{\text{FV}(\mu x. \varphi_1)}) : \text{Int}] \rho',
\]

we have:

\[
[(\Delta), x : \zeta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{m(c_1, d_1)} : \zeta^{\dagger}] (\rho\{x \mapsto w_1\})
\leq_{\zeta'} [(\Delta), x : \zeta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{m(c_2, d_2)} : \zeta^{\dagger}] (\rho'\{x \mapsto w'_1\}).
\]

Let \( m \) and \( m' \) be \([\Delta^{\dagger} \vdash_{\text{st}} \text{exarg}^{(c_1, d_1)}(\Delta_{\text{FV}(\mu x. \varphi_1) y_1 \cdots y_n)}) : \text{Int}] \rho\) and \([\Delta^{\dagger} \vdash_{\text{st}} \text{exarg}^{(c_2, d_2)}(\Delta_{\text{FV}(\mu x. \varphi_1) y_1 \cdots y_n)}) : \text{Int}] \rho'\) respectively. Since \( m \leq m' \), we have:

\[
[(\Delta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{(c_1, d_1)} : \zeta^{\dagger}) \rho]
= (\lambda w_1. [(\Delta^{\dagger}, x : \zeta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{m(c_1, d_1)} : \zeta^{\dagger}] (\rho\{x \mapsto w_1\}))^m(\downarrow_{\zeta'})
\leq_{\zeta'} (\lambda w_1. [(\Delta^{\dagger}, x : \zeta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{m(c_1, d_1)} : \zeta^{\dagger}] (\rho\{x \mapsto w_1\}))^{m'}(\downarrow_{\zeta'})
\leq_{\zeta'} (\lambda w'_1. [(\Delta^{\dagger}, x : \zeta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{m(c_2, d_2)} : \zeta^{\dagger}] (\rho'\{x \mapsto w'_1\}))^{m'}(\downarrow_{\zeta'})
= [(\Delta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{(c_2, d_2)} : \zeta^{\dagger}) \rho'].
\]

We have thus

\[
[(\Delta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{(c_1, d_1)} : \zeta^{\dagger}) \rho] \leq_{\zeta} [(\Delta^{\dagger} \vdash_{\text{st}} \varphi_{1}^{(c_2, d_2)} : \zeta^{\dagger}) \rho']
\]
as required. 

Theorem 3.2 is an immediate corollary of the above lemma.

\textbf{Proof} [Proof of Theorem 3.2] A special case of Lemma \textbf{A.3} where \( \Delta = \emptyset \) and \( \zeta = \ast \).

\[\square\]

\textbf{B More Information on the Experimental Results}

The full results of the experiment for comparison with previous higher-order program verification tools are shown in Table 5 and 6. For all instances, the expected result is “valid.”
Table 5: Results of the experiment for comparison with previous higher-order program verification tools (1/2).

| benchmark   | instance    | result | time   | no. of iter. (prover) | no. of iter. (disprover) | previous tools |
|-------------|-------------|--------|--------|-----------------------|--------------------------|----------------|
| termination | ackermann   | valid  | 136.52 | 2                     | 1                        | verified 5.13  |
| termination | any-down    | valid  | 0.17   | 1                     | 1                        | verified 0.14  |
| termination | append      | valid  | 0.12   | 1                     | 1                        | verified 0.14  |
| termination | binomial    | valid  | 0.20   | 1                     | 1                        | verified 0.64  |
| termination | fibonacci   | valid  | 0.16   | 1                     | 1                        | verified 0.16  |
| termination | foldr       | valid  | 0.13   | 1                     | 1                        | verified 1.23  |
| termination | indirect    | valid  | 0.52   | 1                     | 1                        | verified 1.20  |
| termination | indirectHO | valid  | 0.19   | 1                     | 1                        | verified 8.12  |
| termination | indirectIntro | valid | 0.34   | 1                     | 1                        | verified 27.03 |
| termination | loop2       | valid  | 0.29   | 2                     | 1                        | verified 0.48  |
| termination | map         | valid  | 0.51   | 1                     | 1                        | verified 2.00  |
| termination | mc91        | valid  | 623.44 | 5                     | 3                        | verified 3.57  |
| termination | mult        | valid  | 0.11   | 1                     | 1                        | verified 0.15  |
| termination | nested-loop | valid  | 0.17   | 1                     | 1                        | verified 0.33  |
| termination | partial     | valid  | 0.14   | 1                     | 1                        | verified 1.51  |
| termination | quicksort   | timeout| -      | 5                     | 3                        | timeout        |
| termination | sum         | valid  | 0.12   | 1                     | 1                        | verified 0.12  |
| termination | toChurch    | valid  | 0.13   | 1                     | 1                        | verified 0.61  |
| termination | up-down     | valid  | 0.19   | 1                     | 1                        | verified 0.59  |
| termination | x-plus-2-n  | valid  | 0.17   | 1                     | 1                        | verified 1.79  |
| termination | zip         | valid  | 0.15   | 1                     | 1                        | verified 0.16  |
| non-termination | fib-CPS-nonterm | valid | 0.16   | 1                     | 1                        | verified 0.16  |
| non-termination | fixpoint-nonterm | valid | 0.18   | 1                     | 2                        | verified 0.27  |
| non-termination | foldr-nonterm | valid | 0.28   | 1                     | 2                        | fail           |
| non-termination | indirectHO-e | valid | 0.24   | 1                     | 2                        | verified 0.13  |
| non-termination | indirect-e | valid  | 0.24   | 1                     | 2                        | verified 0.12  |
| non-termination | inf-closure | valid  | 0.25   | 1                     | 1                        | verified 10.66 |
| non-termination | loopHO      | valid  | 0.19   | 1                     | 2                        | verified 1.55  |
| non-termination | passing-cond | valid | 37.15  | 1                     | 2                        | verified 8.30  |
| non-termination | unfoldr-nonterm | valid | 0.13   | 1                     | 1                        | verified 12.15 |
| fair-termination | murase-closure | valid | 0.16   | 1                     | 1                        | verified 12.05 |
| fair-termination | murase-intro | valid  | 0.25   | 1                     | 1                        | verified 11.87 |
| fair-termination | murase-repeat | valid | 0.22   | 1                     | 1                        | verified 2.37  |
| fair-termination | hofmann-2   | valid  | 0.10   | 1                     | 1                        | verified 1.07  |
| fair-termination | koskienen-1 | valid  | 0.14   | 1                     | 1                        | timeout        |
| fair-termination | koskienen-2 | valid  | 0.33   | 2                     | 1                        | verified 3.18  |
| fair-termination | koskienen-3-1 | valid | 0.17   | 1                     | 1                        | verified 2.77  |
| fair-termination | koskienen-3-3 | valid | 0.18   | 1                     | 1                        | verified 5.27  |
| fair-termination | koskienen-4 | valid  | 0.24   | 1                     | 1                        | verified 156.58 |
| fair-termination | lester      | valid  | 0.31   | 1                     | 2                        | timeout        |
Table 6: Results of the experiment for comparison with previous higher-order program verification tools (2/2).

| benchmark                  | MuHFL     | previous tools |
|----------------------------|-----------|----------------|
|                            | result    | no. of iter. (prover) | no. of iter. (disprover) | result | time |
| fair-non-termination       | call-twice| valid           | 0.28                      | 1      | 1    | verified | 1.05 |
| fair-non-termination       | compose   | valid           | 0.13                      | 1      | 1    | verified | 0.86 |
| fair-non-termination       | intro     | valid           | 0.25                      | 2      | 1    | verified | 3.99 |
| fair-non-termination       | loop-CPS  | valid           | 0.12                      | 1      | 1    | verified | 1.55 |
| fair-non-termination       | loop      | valid           | 0.12                      | 1      | 1    | verified | 0.95 |
| fair-non-termination       | murase-closure-buggy | valid | 0.19 | 2 | 1.01 |
| fair-non-termination       | murase-repeat-buggy | valid | 0.20 | 1 | 1.20 |
| fair-non-termination       | nested-if | valid           | 0.35                      | 3      | 1    | verified | 1.47 |
| fair-non-termination       | odd-nonterm | valid        | 0.23                      | 2      | timeout - |
| fair-non-termination       | op-loop   | valid           | 0.19                      | 2      | 1    | verified | 1.63 |
| fair-non-termination       | update-max | valid          | 0.43                      | 3      | 1    | verified | 1.11 |
| fair-non-termination       | update-max-CPS | valid | 0.35 | 3 | 1.76 |
| fair-non-termination       | koskinen-1-buggy | valid | 0.14 | 1 | 3.55 |
| fair-non-termination       | koskinen-2-buggy | valid | 0.16 | 1 | 4.40 |
| fair-non-termination       | koskinen-3-1-buggy | valid | 0.09 | 1 | 2.07 |
| fair-non-termination       | koskinen-3-3-buggy | valid | 0.21 | 2 | 9.34 |
| termination-ho             | ackermann-ho | timeout       | -                         | 4      | 3    | fail    - |
| termination-ho             | any-down-ho | valid          | 0.36                      | 1      | 1    | timeout - |
| termination-ho             | append-ho  | valid           | 0.54                      | 1      | 1    | timeout - |
| termination-ho             | binomial-ho | valid          | 0.95                      | 1      | 1    | fail    - |
| termination-ho             | fibonacci-ho | valid       | 1.04                      | 1      | 1    | timeout - |
| termination-ho             | foldr-ho   | timeout        | -                         | 4      | 8    | timeout - |
| termination-ho             | indirect-ho | valid          | 0.59                      | 1      | 1    | timeout - |
| termination-ho             | indirectHO-ho | valid       | 0.29                      | 1      | 1    | timeout - |
| termination-ho             | indirectIntro-ho | timeout | -      | 5      | 3    | timeout - |
| termination-ho             | loop2-ho   | timeout        | -                         | 5      | 3    | timeout - |
| termination-ho             | map-ho     | timeout        | -                         | 4      | 9    | timeout - |
| termination-ho             | mc91-ho    | timeout        | -                         | 4      | 3    | timeout - |
| termination-ho             | mult-ho    | valid           | 0.67                      | 1      | 1    | timeout - |
| termination-ho             | nested-loop-ho | valid     | 0.64                      | 1      | 1    | timeout - |
| termination-ho             | partial-ho | valid           | 0.32                      | 1      | 1    | timeout - |
| termination-ho             | quicksort-ho | timeout       | -                         | 3      | 3    | timeout - |
| termination-ho             | sum-ho     | valid           | 1.05                      | 1      | 1    | timeout - |
| termination-ho             | toChurch-ho | timeout        | -                         | 5      | 9    | timeout - |
| termination-ho             | up-down-ho | timeout        | -                         | 5      | 3    | timeout - |
| termination-ho             | x-plus-2-n-ho | timeout     | -                         | 5      | 3    | timeout - |
| termination-ho             | zip-ho     | valid           | 0.60                      | 1      | 1    | timeout - |
| fair-termination-ho        | murase-closure-ho | valid    | 0.68 | 1 | timeout - |
| fair-termination-ho        | murase-intro-ho | valid    | 0.43 | 1 | timeout - |
| fair-termination-ho        | murase-repeat-ho | valid  | 2.02 | 1 | timeout - |
| fair-termination-ho        | hofmann-2-ho | valid        | 0.15                      | 1      | 1    | verified | 3.25 |
| fair-termination-ho        | koskinen-1-ho | timeout       | -                         | 4      | 8    | fail    - |
| fair-termination-ho        | koskinen-2-ho | timeout       | -                         | 3      | 3    | timeout - |
| fair-termination-ho        | koskinen-3-1-ho | timeout  | -                         | 6      | 3    | timeout - |
| fair-termination-ho        | koskinen-3-3-ho | valid     | 0.64 | 1 | timeout - |
| fair-termination-ho        | koskinen-4-ho | valid        | 1.10                      | 1      | 1    | verified | 6.33 |
| fair-termination-ho        | lester-ho  | valid           | 0.70                      | 1      | 1    | timeout - |