Temporal Verification of Higher-Order Functional Programs

Akihiro Murase *  
Nagoya University, Japan  
murase@sqlab.jp

Tachio Terauchi  
JAIST, Japan  
terauchi@jaist.ac.jp

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

Ryosuke Sato  
University of Tokyo, Japan  
ryosuke@kb.is.s.u-tokyo.ac.jp

Hiroshi Unno  
University of Tsukuba, Japan  
uhiro@cs.tsukuba.ac.jp

Abstract

We present an automated approach to verifying arbitrary omega-regular properties of higher-order functional programs. Previous automated methods proposed for this class of programs could only handle safety properties or termination, and our approach is the first to be able to verify arbitrary omega-regular liveness properties.

Our approach is automata-theoretic, and extends our recent work on binary-reachability-based approach to automated termination verification of higher-order functional programs to fair termination published in ESOP 2014. In that work, we have shown that checking disjunctive well-foundedness of the calling relation is sound and complete for termination. The extension to fair termination is tricky, however, because the straightforward extension that checks disjunctive well-foundedness of the fair calling relation turns out to be unsound, as we shall show in the paper. Roughly, our solution is to check fairness on the transition relation instead of the calling relation, and propagate the information to determine when it is necessary and sufficient to check for disjunctive well-foundedness on the calling relation. We prove that our approach is sound and complete. We have implemented a prototype of our approach, and confirmed that it is able to automatically verify liveness properties of some non-trivial higher-order programs.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Formal methods, Model checking; F.3.1 [Logics and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs— Assertions, Invariants, Mechanical verification; F.3.2 [Logics and Meaning of Programs]: Semantics of Programming Languages—Program analysis; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Temporal logic

Keywords Automatic Verification, Temporal Properties, Higher-Order Programs, Automata-Theoretic Verification, CEGAR, Binary Reachability Analysis

1. Introduction

Recent years have witnessed a significant progress in techniques for automatically verifying higher-order functional (or, procedural) programs. Besides the purely theoretical interests, the progress is motivated in part by the desire to verify real world programs that contain higher-order functions (e.g., programs written in functional languages such as OCaml [15, 19, 29, 34–36] and Haskell [38, 39]), and also in part by the observation that higher-order functions can be used to (often concisely) model advanced programming language features, such as recursive data structures, exceptions, and continuations, thus turning the verifiers for higher-order functional programs into verifiers for programs containing such features [30].

For the case the data is finite and the program is well-typed, Ong and others [9, 18, 23–25] have presented algorithms for verifying arbitrary modal-µ temporal properties of higher-order functional programs. However, for the infinite data case, the current state-of-the-art methods are limited to either only safety properties [15, 19, 29, 34–36, 38, 39] or termination [10, 17, 22, 31]. This is in stark contrast to imperative programs which have appropriated effective automated techniques for verifying an expressive range of temporal properties [2–5]. To rectify the situation, this paper presents the first approach to automatically verifying arbitrary ω-regular temporal properties of (possibly untyped) infinite data higher-order functional programs. (Recall that an ω-regular property is a set of infinite words recognized by a Büchi automaton, and includes any property expressible by linear temporal logic.) Next, we give an informal overview of our approach.

1.1 Informal Overview

Our approach follows the automata-theoretic framework [37], and reduces the temporal property verification to verifying that the program has no infinite fair execution traces (i.e., fair termination). To verify fair termination for higher-order functional programs, we build on our recent work [22] on (plain) termination verification for higher-order functional programs. Next, we briefly overview the main ideas of [22]. We abbreviate a sequence \( w_1, \ldots, w_k \) to \( \bar{w} \) and write \( |\bar{w}| \) for the length of the sequence \( \bar{w} \) below.

The approach taken in [22] adopts the binary reachability analysis (BRA) method that was originally proposed by Cook et al. for termination verification of imperative programs [6]. Recall that a binary relation is said to be disjunctively well-founded if it is a finite union of well-founded relations. Let \( \text{Trans} \) be the transition relation...
relation of a program $P$ (i.e., $(e,e') \in Trans_P$ if and only if $e$ and $e'$ are states occurring in an execution of $P$ such that there is a transition from $e$ to $e'$). $P$ is terminating if and only if $Trans_P^+$ is disjunctively well-founded [28] (we write $R$ for the transitive closure of the binary relation $R$), and [6] presents an automatic method, the BRA approach, which checks the latter by an iterative reduction to plain reachability verification problems.

As remarked in previous literature on BRA [6, 22], advantages of the BRA approach over the other approaches to termination verification are that termination arguments can be flexibly adjusted for each program, and that precise flow information can be taken into account in the plain reachability verification phase. The latter advantage is particularly important since the termination property often depends on safety properties.

Whereas BRA for imperative programs checks disjunctive well-foundedness of the transitive closure of the transition relation, the key idea in [22] is to check that of the calling relation. More formally, a calling relation $Call_P$ is the set of pairs $(f\,\overline{w}, g\,\overline{v})$ where $f\overline{w}$ and $g\overline{v}$ are total applications that occur in an execution of $P$ such that the call $g\,\overline{v}$ is made from $f\overline{w}$. A key result proved in [22] is that $P$ is terminating if and only if $Call_P^+$ is disjunctively well-founded. (We underscore that the calling relation is over the method proposed by [6, 28] is language agnostic because any termination even for higher-order functional programs (i.e., the program can be considered a transition system). The advantage of using the calling relation instead of the transition relation is that it avoids the need to explicitly reason about the change in the calling context (e.g., a program terminates (i.e., disjunctively well-founded relation containing the transitive closure of the calling relation) sufficient to prove termination: $D_1 \cup D_2$ where

$$D_1 = \{(ac\,m\,n, ac\,m'\,n') \mid m > m' \land m' \geq 0\}$$

$$D_2 = \{(ac\,m\,n, ac\,m'\,n') \mid n > m' \land n' \geq 0\}$$

Indeed, $Call_P^+ \subseteq D_1 \cup D_2$ because any (transitive) recursive call to ac either decreases on the first argument or the second argument. However, the transition relation of the program is quite complex, as seen below in the transition sequence given $m, n > 0$.

$$ac\,m\,n \rightarrow ac\,(m-1)\,(ac\,(m-1)\,ac\,m\,(n-1))$$

$$\vdots$$

$$ac\,(m-1)\,(ac\,(m-1)\,(ac\,(m-1)\,0)\ldots)$$

$$ac\,(m-1)\,(ac\,(m-1)\,(ac\,(m-1)\,0)\ldots)$$

Consequently, the required termination argument also becomes more complex. In general, the complexity comes from the need to reason about changes in the calling context (e.g., a program terminates because the number of pending callers on the call stack decreases), and hence avoiding such a reasoning by using the calling relation instead of the transition relation is crucial to practical verification.²

We remark that the above example is only first-order, and that things become even more intricate when we move to higher-order. A contribution of [22] shows that checking disjunctive well-foundedness on the transitive closure of the calling relation is sound and complete even in the presence of higher-order functions.

**Extending to Fair Termination.** In this work, we extend the approach of [22] to fair termination. Recall that an infinite sequence $\pi$ is said to be fair with respect to a Streett fairness constraint $C = \{(p_1, q_1), (p_2, q_2), \ldots, (p_n, q_n)\}$ if for each $(p_i, q_i) \in C$, either $p_i$ is true for only finitely many elements of $\pi$ or $q_i$ is true for infinitely many elements of $\pi$. $P$ is said to be fair terminating under $C$ if $P$ has no infinite execution trace that is fair with respect to $C$.

We call a finite sequence $\pi$ fair with respect to $C$ if for each $(p_i, q_i) \in C$, either $p_i$ is false for all elements of $\pi$ or $q_i$ is true for some element of $\pi$. For a binary relation $R$, let $R^{fair}_{\pi C}$ be the subset of $R^+$ that only considers the sequences of transitions that are fair with respect to $C$. Using a result in a paper by Pnueli et al. [26], Cook et al. [2] have shown that $P$ is fair terminating under $C$ if and only if $Trans_{fair R C}$ (i.e., $R_{\pi C}^{fair}$ with $R = Trans_P$) is disjunctively well-founded. The latter result is then used to obtain a BRA approach to sound and complete automatic fair-termination verification of imperative programs.

knowing the result for the plain termination case shown by [22], one may be lead to believe that the result of [22] can be applied directly to obtain the following scheme for checking fair termination of higher-order functional programs: check disjunctive well-foundedness of fair sequences of calling relations. That is, check that $Call_P^{fair C}$ is disjunctively well-founded. However, perhaps somewhat surprisingly, the approach turns out to be unsound.³ To see this, consider the program $P_{ac}$ that starts by calling the following function $f$ with a positive argument:

$$f\,x = \text{if } x \leq 0 \text{ then } 0 \text{ else } (f\,0); (f\,1)$$

Let $C = \{(true, f\,0)\}$, that is, $f$ is called with the argument 0 infinitely often in an execution that is fair with respect to $C$. Clearly, $P_{ac}$ is not fair terminating under $C$ since any execution of $P_{ac}$ is non-terminating and contains infinitely many calls to

² In fact, the set of functional programs that can be proved terminating by disjunctive well-foundedness on the transitive closure of the calling relation is strictly larger than the set that can be proved by that of the transition relation. for a class of termination arguments such as linear ranking functions [1, 40].

³ For simplicity, this section restricts state predicates to function applications. Section 2 introduces more general events that can be used to express arbitrary $\omega$-regular temporal properties of program states. A similar issue applies in that setting.
f with the argument 0. Nonetheless, \( \text{Call}_{P_c}^{\to f} \) is disjunctively well-founded. Indeed, it can be seen from Figure 1 which shows the graph of \( \text{Call}_{P_c} \) (i.e., \( P_c \)'s call tree [22]) that any infinite path in \( \text{Call}_{P_c} \) is unfair with respect to \( C \).

The solution we propose in this paper is to decide when a transition sequence is fair by inspecting the transition relation, and use that information to decide when we should check for disjunctive well-foundedness of the calling relation. More formally, let \( \rightarrow_P \) be the one-step transition of \( P \) and \( \mathcal{R}_P \) be the set of states reachable from an initial state of \( P \). Note that we have:

\[
\begin{align*}
\text{Trans}_P^+ &= \{(e, e') \mid e \in \mathcal{R}_P \land e \rightarrow_P e'\} \\
\text{Trans}^{\to f}_P &= \{(e, e') \mid e \in \mathcal{R}_P \land e \rightarrow_P \text{Call}_{P_c}^{\to f} e'\} \\
\text{Call}_P &= \{(f \bar{w}, g v) \mid E[f \bar{w}] \in \mathcal{R}_P \land f \bar{w} \rightarrow_P g v \land \text{arity}(f) = |\bar{w}| \land \text{arity}(g) = |v|\}
\end{align*}
\]

where \( E \) and \( E' \) are evaluation contexts. Let the relation \( \triangleright_P^C \) be defined as follows.

\[
\triangleright_P^C = \{(f \bar{w}, g v) \mid E[f \bar{w}] \in \mathcal{R}_P \land f \bar{w} \rightarrow_P \text{Call}_{P_c}^{\to f} E'[g v] \land \text{arity}(f) = |\bar{w}| \land \text{arity}(g) = |v|\}
\]

That is, \( \triangleright_P^C \) is the above characterization of \( \text{Call}_P^{\to f} \) but with \( \rightarrow_P \) replaced by \( \rightarrow_P \text{Call}_{P_c}^{\to f} \). The main result of the paper (Theorem 3.1) shows that \( P \) is fair terminating under \( C \) if and only if \( \triangleright_P^C \) is disjunctively well-founded. Thus, the verification method we propose in this paper checks for the disjunctive well-foundedness of \( \triangleright_P^C \). As with the work on plain termination which checks the disjunctive well-foundedness of \( \text{Call}_P^{\to f} \), this is important advantage of this approach is that it avoids the explicit reasoning about changes in the calling context.

As an example, recall the call program \( P_{\text{ex}} \) from above. Let \( C = \{(\text{true}, f 0)\} \) again. Recall that \( P_{\text{ex}} \) is not fair terminating under \( C \). Indeed, \( \triangleright_P^{C} \) is not disjunctively well-founded because there is an infinite sequence:

\[
f n \triangleright_P^{C} f 1 \triangleright_P^{C} f 1 \triangleright_P^{C} \cdots
\]

This follows because \( f 1 \rightarrow_P P_{\text{ex}} (f 0); (f 1) \) and \( (f 0); (f 1) \rightarrow_P P_{\text{ex}} f 1, \) and therefore \( f 1 \rightarrow_P P_{\text{ex}} f 1 \).

As a fair terminating example, consider \( P_{\text{ex}} \) again, but this time with respect to the fairness constraint \( C' = \{(f 0, \text{false})\} \). That is, \( f \) is called with 0 only finitely often in a trace fair with respect to \( C' \). \( P_{\text{ex}} \) is fair terminating under \( C' \), because any infinite execution of \( P_{\text{ex}} \) is unfair, i.e., it contains infinitely many 0 calls. We can show that \( \triangleright_P^{C'} \) is disjunctively well-founded. In fact, \( \triangleright_P^{C'} = \emptyset \) because \( f n \not\triangleright_P^{C'} f \) for \( i \) and \( f 1 \not\triangleright_P^{C'} f 1 \) for \( i \) in \( \{0, 1\} \).

As also done in [22], we take advantage of the fact that disjunctive well-foundedness of a calling relation can be checked per function basis. That is, let \( f_1, \ldots, f_n \) be the functions defined in \( P \). Then, \( \triangleright_P^C \) is disjunctively well-founded if and only if for each \( f_i, \) \( \text{REC}_{P, C}(f_i) = \{((\bar{w}, \bar{v}), (\bar{w}, \bar{v})) \mid f_i \triangleright_P^C (\bar{w}, \bar{v}) \} \) is disjunctively well-founded.

**Overall Flow of Verification.** As in previous work on plain or fair termination verification via BRA [2, 6, 22], we solve the disjunctive well-foundedness checking problem in a counterexample-guided manner. That is, we start with some candidate termination argument (i.e., disjunctively well-founded relation) \( D \). Then, we check if the candidate is an actual termination argument (i.e., whether \( \text{REC}_{P, C}(f_i) \subseteq D \) holds) by a reduction to reachability checking, and if not, we update the candidate via a counterexample analysis and repeat.

Figure 2 shows the overview of the overall verification process. Step 1 is a program transformation that reduces the candidate termination argument checking problem soundly and completely to reachability checking, and is a key technical contribution of the paper. The step builds a transformed program \( [P]_{f, D, C} \) that is assertion safe if and only if \( \text{REC}_{P, C}(f_i) \subseteq D \) (Theorem 4.1).

The program transformation adopts ideas from those proposed in the previous work [2, 22]. Namely, it uses the idea from [2] to detect when a sequence of transitions is fair, and uses the idea from [22] to check that the calling relation over such sequences is contained in the candidate \( D \). Section 4 describes the formal details of the program transformation. The reachability checking (Step 2) is standard, and we refer to the previous work [19, 34, 36] for details. We use the approach proposed in [22] for the candidate termination argument inference (Step 3), and we give a brief overview of the process below. We have implemented a prototype of our verification method, and we show that it is able to verify non-trivial liveness properties of higher-order functional programs.

In summary, our contributions are: (i) The first sound and complete approach to verification of arbitrary \( \omega \)-regular properties of infinite data higher-order functional programs. Our approach combines and extends ideas from previous work on plain termination verification for high-order functional programs [22] and fair termination verification for imperative programs [22] in a non-trivial way. (ii) Experiments with a prototype implementation to show the effectiveness of our approach.

**On Soundness and Completeness.** As in previous work on BRA, the soundness and completeness of our approach are relative to the soundness and completeness of the backend reachability checking and candidate termination argument inference. More precisely, our approach is sound if the reachability checking is sound and only disjunctively well-founded relations are inferred as candidate termination arguments, and it is complete if the reachability checking is complete and the candidate termination argument inference always infers some relation given a spurious counterexample.

We note that the relative soundness and completeness are thanks to the fact that the BRA approach reduces the problem of verifying (plain or fair) termination to that of verifying plain reachability without any loss of precision. As remarked before, this is important for verification in practice as termination properties often depend on safety properties.

**Termination Argument Inference.** We give a brief overview of the termination argument inference process. A counterexample returned by the reachability checker (cf. Step 2 of Figure 2) is a path of the transformed program \( [P]_{f, D, C} \) from the initial state to an assertion error. That is, it is a finite sequence of (symbolic) executions \( \pi \) of \( [P]_{f, D, C} \) such that \( \{(\bar{w}, \bar{v}) \mid [\pi](\bar{w}, \bar{v})\} \not\subseteq D \) where...
\[ \Box \] is the strongest post condition of \( \Box \). (We refer to a previous work [19] for details on how counterexamples can be generated in a reachability checker for higher-order functional programs.)

By the soundness of the program transformation (Theorem 4.1), this implies that
\[
\{(\overline{w}, \overline{v}) \mid \Box (\overline{w}, \overline{v}) \subseteq \text{REC}_{P,C}(f) \subseteq D
\]
assuming that the reachability checker is sound. Then, via a constraint-based ranking function inference method [7, 27], we infer a disjunctively well-founded relation \( D' \) such that \( \{(\overline{w}, \overline{v}) \mid \Box (\overline{w}, \overline{v}) \subseteq D' \). If no such a relation is found, 4 then we halt the verification process by outputting “P may not fair terminate under \( C' \).” Otherwise, we obtain \( D \cup D' \) as the updated candidate termination argument to be given to Step 1 of Figure 2, and continue the verification process.

As also observed in [22], higher-order functional programs sometimes require termination arguments that are over function values. However, the ranking function inference methods are typically restricted to first-order values (e.g., only inferring linear ranking functions over integers). Hence, to handle such a case, [22] proposes an approach where first-order implicit parameters are added to the (transformed) program so that higher-order termination arguments can be expressed as those over the implicit parameters. The technique can be adopted straightforwardly to this work’s setting to handle higher-order termination arguments.

Disproving Fair Termination. As in previous work on plain or fair termination verification via BRA [2, 6, 22], our method is not suited for disproving fair termination, i.e., for showing that there is an infinite fair execution sequence. (Assuming the reachability checker and the termination argument inference are complete), our method is able to prove simple cases of fair non-termination, such as
\[
f_1 \rightarrow_{\text{fair}_{P}} f_2 \rightarrow_{\text{fair}_{P}} f_3 \rightarrow_{\text{fair}_{P}} \ldots,
\]
for the case, \( \text{REC}_{P,C}(f) \) contains (1, 1). Our method cannot, however, detect a non-looping non-termination, like
\[
f_1 \rightarrow_{\text{fair}_{P}} f_2 \rightarrow_{\text{fair}_{P}} f_3 \rightarrow_{\text{fair}_{P}} \ldots.
\]

Given a program that contains such a non-looping infinite execution sequence, the counterexample-guided refinement loop of Figure 2 would diverge (assuming the reachability checker is sound and complete and the termination argument inference is complete). For disproving fair termination of higher-order programs, we can extend the recently proposed method for disproving plain termination of higher-order programs [21]. We will discuss it in a separate paper, since the method is quite different from and orthogonal to the one discussed in the present paper.

Paper Organization. The rest of the paper is organized as follows. We discuss related work next. Section 2 presents the target programming language and relevant preliminary concepts. Sections 3 and 4 contain the formal details of the reduction from fair termination to reachability checking which is the main technical contribution of the paper. Section 3 formalizes the characterization of fair termination for higher-order functional programs as checking disjunctive well-foundedness of \( \text{REC}_{P,C}(f) \) (i.e., checking \( \text{REC}_{P,C}(f) \) is disjunctively well-founded for every \( f \) in \( P \)), and Section 4 describes the program transformation that reduces the problem of checking if \( \text{REC}_{P,C}(f) \) is contained in the given candidate termination argument to reachability checking. Section 5 reports on a preliminary implementation and experiment results, and Section 6 concludes the paper.

1.2 Related Work
To our knowledge, this paper is the first to propose a sound and complete approach to automatically verifying arbitrary \( \omega \)-regular properties of infinite data higher-order functional programs. As remarked before, previous work on automated methods for higher-order functional programs are limited to (typed) finite data programs [9, 18, 23–25], or termination [10, 17, 22, 31] and safety properties [15, 19, 29, 34–36, 38, 39] for infinite data programs.

A notable exception to the above is the work by Skalka et al. [32, 33] that proposes a type-and-effect system for verifying arbitrary \( \omega \)-regular properties of higher-order functional programs. In their approach, a type-and-effect system is used as a static analysis to obtain a sound finite-state abstraction of the program. Then, a model checker (for finite state systems) is used to check the abstraction against the given temporal property. Because of the over-approximation, the approach is incomplete and somewhat imprecise. In particular, any recursive call will be abstracted as an unbounded loop which makes it difficult to be used for deducing non-trivial liveness properties.

Related to this work is the recent work by Hofmann and Chen [13] that proposes a similar type-and-effect system. Their work shows a close correspondence between the type-and-effect system and automata theoretic concepts, and shows that the system is sound and complete for \( \omega \)-regular property verification of simple finite data first-order functional programs without conditional branches. It is unclear if the system can be extended to higher-order functions (the paper briefly sketches such an extension) or infinite data.

In a recent work [20], Koskinen and Terauchi have proposed a framework for verifying arbitrary \( \omega \)-regular properties of infinite data higher-order functional programs in a compositional manner. However, their paper does not discuss automation, and moreover, the approach requires an external “oracle” verifier to deduce non-trivial liveness properties (the approach is sound and complete relative to the soundness and completeness of the oracles). Our work is complementary to theirs in the sense that the verifier proposed here can be put to use as an oracle in their framework.

2. Source Language
The sets of expressions and values, ranged over by \( e \) and \( v \) respectively, are defined by the grammar shown in Figure 3. Here, \( c \) ranges over the set of constants (including integers and the unit value \( 0 \)), \( \text{op} \) ranges over the set of binary operators, and \( x \) over the set of variables. We write \( \text{true} \) and \( \text{false} \) for 0 and 1 respectively. We write \( \text{[op]} \) for the semantics of the operator \( \text{op} \). The value \( \langle x, v_1, \ldots, v_n \rangle \) expresses a function closure; it occurs only at run-time. We write \( \langle v_1 / x_1, \ldots, v_n / x_n \rangle \) for the capture-avoiding parallel substitution of \( v_1, \ldots, v_n \) for \( x_1, \ldots, x_n \).

In the formal syntax of expressions above, we restrict the positions where effectful expressions may occur. For readability, we often write (especially in examples) if \( e_0 \) then \( e_1 \) else \( e_2 \) for let \( x = e_0 \) in if \( x \) then \( e_1 \) else \( e_2 \), and if \( e_1 \) \( \cdot \cdot \cdot \) \( e_n \) for let \( x_1 = e_1 \) in \( \cdot \cdot \cdot \) let \( x_n = e_n \) in \( f x_1 \cdot \cdot \cdot x_n \). We also write \( e_1; e_2 \) for let \( x = e_1 \) in \( e_2 \) when \( x \) does not occur in \( e_2 \).
The program repeatedly calls

Example 2.2

We write

\[ E[\text{repeat } g = \text{let } x = \text{rand}(1) \text{ in } g x] \]

The program repeatedly calls \( f \) with a random integer value as an argument. Since each call of \( f \) eventually raises an event \( A \), the program never terminates and raises \( A \) infinitely often.

A (Streett) fairness constraint \( C \) is a set of event pairs:

\( \{(A_1, B_1), \ldots, (A_n, B_n)\} \)

Intuitively, it describes the fairness constraint that if \( A_i \) occurs infinitely often, \( B_i \) must also happen infinitely often.

Remark 2.1 Usually, a fairness constraint is of the form

\( \{(p_1, q_1), \ldots, (p_n, q_n)\} \)

where \( p_i, q_i \) are sets of states, meaning “whenever one of the states in \( p_i \) is visited infinitely often, one of the states in \( q_i \) must also be visited infinitely often.” In our language, a single event \( A_p \) can be used to express the property that one of the states in \( p \) is visited; hence, a fairness constraint has been defined above as a set of pairs of events, rather than as a set of pairs of “sets of” events.

Definition 2.1 Let \( C \) be \( \{(A_1, B_1), \ldots, (A_n, B_n)\} \), and \( \sigma \) be a possibly infinite sequence of events. We write \( \sigma \models C \) when, for every \( i \in \{1, \ldots, n\} \), \( A_i \) occurs infinitely often in \( \sigma \), so does \( B_i \).

Definition 2.2 Let \( C \) be \( \{(A_1, B_1), \ldots, (A_n, B_n)\} \). A program \( P \) is fair terminating under \( C \) if, for every infinite reduction sequence

main \( \xrightarrow{} \) \( e_1 \xrightarrow{} e_2 \xrightarrow{} e_3 \xrightarrow{} \cdots \),

\( \ell_1 \ell_2 \ell_3 \cdots \models C \) does not hold.

In other words, a program is fair terminating if there is no infinite fair reduction sequence.

Vardi [37] has shown that verification of arbitrary \( \omega \)-regular temporal properties can be reduced to that of fair termination. We give below some examples of reductions. Here we assume that the body of every function definition starts with \( \text{let } x = v \text{ in } e \) to capture the event that \( f \) takes a unit value and (if terminates) returns a unit value.

Example 2.1 The example on the function \( f \) in Section 1 is expressed as the following program \( P_a \):

\[
\begin{align*}
f(x) = & \text{if } x < 0 \text{ then } \ast \text{ else } \\
& \text{if } x = 0 \text{ then } \text{event } A \text{ else } (f 0); (f 1) \\
\text{main } y & = \text{let } x = \text{rand}(1) \text{ in } f x
\end{align*}
\]

Here, we have inserted event \( A \) to capture the event that \( f 0 \) is called.

Example 2.2 Let \( P_{\text{rep}} \) be the program consisting of the following function definitions.

\[
\begin{align*}
\text{repeat } g & = \text{let } x = \text{rand}(1) \text{ in } g x; \text{repeat } g \\
f(x) & = \text{if } x > 0 \text{ then } f(x - 1) \text{ else } \text{event } A \\
\text{main } y & = \text{repeat } f
\end{align*}
\]

The program repeatedly calls \( f \) with a random integer value as an argument. Since each call of \( f \) eventually raises an event \( A \), the program never terminates and raises \( A \) infinitely often.

\[ \text{let } x = \text{rand}(1) \text{ in } x \]

\( E[\text{rand}()] \xrightarrow{} E[n] \)

\[ E[v_1 \text{ op } v_2] \xrightarrow{} E[v] \]

\[ E[\text{if } e \xrightarrow{} E[[e]] \text{ if } e \xrightarrow{} E[[e]] \text{ then } \text{event } \ell \xrightarrow{} E[[\ell]] \]

\[ E[\text{let } x = v \text{ in } e] \xrightarrow{} E[[e/x]] \]

The set of evaluation contexts, ranged over by \( E \), is given by:

\[ E ::= [\ ] | \text{let } x = E \text{ in } E \]

The labeled reduction \( e \xrightarrow{\ell} e' \), where \( \ell \) is either an event or an empty sequence, is defined by the rules shown in Figure 4. We often omit \( E \) and \( e \) in the reduction relation.

We write \( \xi_1, \xi_2, \ldots \xrightarrow{p}^* \) for the relational composition \( \xi_1 \xrightarrow{p} \xi_2 \xrightarrow{p} \cdots \xrightarrow{p} \xi_n \).

Let \( \sigma = \ell_1 \ell_2 \ell_3 \cdots \) be a possibly infinite sequence of labels. We write \( e \xrightarrow{\sigma} \) if there exists a (possibly infinite) sequence of expressions \( e_1, e_2, \ldots \) such that

\[ e \xrightarrow{\ell_1} e_1 \xrightarrow{\ell_2} e_2 \xrightarrow{\ell_3} e_3 \cdots \]

with \( \sigma = \ell_1 \ell_2 \ell_3 \cdots \).

Figure 4. Source language: reduction relation
• Let $P'$ be the program obtained from a program $P$ by replacing the definition of the main function: $\text{main } x = e$ with
  \[ \text{main } x = e; \text{loop } \ast \]
  \[ \text{loop } x = \text{event } \text{Never}; \text{loop } x \]
Then $P'$ is fair terminating under \{\{\text{Call1}, \text{Never}\}\} if and only if $P$ never terminates.

**Remark 2.2** In general, given the program $P$ and the $\omega$-regular temporal property $\phi$ to be verified, the automata-theoretic approach [37] builds an $\omega$-automaton $A_{\neg \phi}$ that recognizes the complement of $\phi$, and verifies that the product program $P \times A_{\neg \phi}$ has no infinite runs. This is a fair-termination problem because the fairness constraint from $A_{\neg \phi}$ is carried over to the product program.

Note that the use of Strengthen fairness constraints in our work stipulates that $A_{\neg \phi}$ is represented by a Strengthen automaton. Strengthen automata are preferred over Büchi automata in our setting because a non-deterministic Büchi automaton is required to express an $\omega$-regular property in general which complicates our approach.

**Remark 2.3** Our source language is untyped. Therefore, a program evaluation may get stuck, and we consider a reduction sequence that ends with a stuck expression as terminating. Our approach is sound and complete even for untyped languages. But, our implementation currently supports only the typed subset because it uses a higher-order program model checker for a typed language [19] as the backend reachability checker.

## 3. Fair Termination Verification via (Fair) Binary Reachability

This section reduces the verification of fair termination to that of a binary reachability problem. Our notion of binary reachability, which we call fair binary reachability, is an extension of the one we proposed for plain termination verification [22]. Unlike in the standard binary reachability for imperative programs [28], we track only the relationship between the arguments of function calls in the calling context (cf. Section 1.1).

**Definition 3.1** Let $\sigma$ be a sequence of events, and let $C = \{ (A_1, B_1), \ldots, (A_n, B_n) \}$ be a fairness constraint. We write $\sigma \models_{\text{fair}} C$ if, for every $i \in \{1, \ldots, n\}$, either (i) $A_i$ does not occur in $\sigma$ or (ii) $B_i$ occurs in $\sigma$.

**Definition 3.2** Let $P$ be a program, $f$ a function defined in $P$, and $C$ a fairness constraint. We write $f \vdash_C^g \tilde{w}$ if there exist $\sigma_1, \sigma_2, E_1, E_2$ such that: (i) $\text{main } \vdash_{\ast} E_1[f \tilde{w}]$, (ii) $f \tilde{w} \vdash_{\ast} E_2[g \tilde{w}]$, (iii) $\sigma_2 \models_{\text{fair}} C$, and (iv) $\text{arity}(f) = |\tilde{w}|$ and $\text{arity}(g) = |\tilde{w}|$. We write $\text{REC}_P(f)$ for the set
  \[ \{ (\tilde{w}, \tilde{v}) \mid f \tilde{w} \vdash_C^g f \tilde{v} \}, \]
and call it a fair recursion relation.

The following is the key theorem for our verification method.

**Theorem 3.1** A program $P$ is fair terminating under $C$, if and only if, $\text{REC}_P(f)$ is well-founded (i.e., there is no infinite chain $\tilde{v}_0 \vdash_{\ast} \text{REC}_P(f) \vdash_{\ast} \text{REC}_P(f) \vdash_{\ast} \text{REC}_P(f) \vdash_{\ast} \cdots$) for every function $f$ defined in $P$.

Since the relation $\text{REC}_P(f)$ is transitive, $\text{REC}_P(f)$ is well-founded if and only if $\text{REC}_P(f)$ is disjunctively well-founded (i.e., is a finite union of well-founded relations) [28]. Thus, to show that $P$ is fair terminating under $C$, it suffices to pick a disjunctively well-founded relation $D_f$ for each function $f$, and show that $\text{REC}_P(f) \subseteq D_f$. We call the problem of proving $\text{REC}_P(f) \subseteq D_f$ a fair binary reachability problem. How to solve the fair binary reachability problem is discussed in Section 4.

**Example 3.1** Recall Example 2.2. To check that $P_{\text{rep}}$ raises events $A$ infinitely often, it suffices to verify
  \[ \text{REC}_P(f) \subseteq D_f \quad \text{REC}_P(f) \subseteq D_{\text{main}} \quad \text{REC}_P(f) \subseteq D_{\text{repeat}} \]
for $C = \{ (A, \text{Never}) \}$, $D_f = \{ (m, n) \mid m > n \geq 0 \}$, and $D_{\text{main}} = D_{\text{repeat}} = \emptyset$. Note that $\text{REC}_P(f)$ (repeat) $= \emptyset$ because, whenever $f$ $\vdash_{\ast} E[\text{repeat } f]$, $\sigma$ contains $A$, so that $\sigma \not\models_{\text{fair}} C$.

The rest of this section is devoted to the proof of Theorem 3.1, which is divided into Theorems 3.2 and 3.3 below.

**Theorem 3.2** (Soundness) If $\text{REC}_P(f)$ is well-founded for every function $f$ in $P$, then $P$ is fair terminating.

**Proof** The proof is by contradiction. Suppose that $\text{REC}_P(f)$ is well-founded for every function $f$ in $P$, but $P$ is not fair terminating, i.e., there exists an infinite reduction sequence $\pi$ that is fair. Since $\pi$ is an infinite sequence, there must be a function $f$ such that $\pi$ can be decomposed to:
  \[ \text{main } \vdash_{\ast} e_1 \vdash_{\ast} e_2 \vdash_{\ast} e_3 \vdash_{\ast} \cdots \]
where $e_i = E_1[\cdots E_i[f \tilde{v}_i] \cdots]$ and $f \tilde{v}_i \vdash_{\ast} E_{i+1}[f \tilde{v}_{i+1}]$ [22]. Let $C = C_1 \cup C_2$, where (i) each event in $\{A \mid (A, B) \in C_1\}$ occurs only finitely often, and (ii) each event in $\{A \mid (A, B) \in C_2\}$ occurs infinitely often in $P$. Let $P_i$ and $Q_i$ be $\{A \mid (A, B) \in C_i\}$ and $\{B \mid (A, B) \in C_i\}$ respectively. As $\pi$ is fair, there exists $k$ such that, in $\sigma_{k+1} \cdots$, no event in $P_k$ occurs and every event in $Q_k$ occurs infinitely often. Now, let us define $n_1, n_2, \ldots$ inductively by: (i) $n_1 = k$, and (ii) $n_{i+1}$ is the least $j > n_i$ such that every event in $Q_j$ occurs in $\sigma_{n_1 \sigma_{n_1+1} \cdots \sigma_{j-1}}$. Then, we have an infinite sequence:
  \[ f \tilde{v}_{n_1} \vdash_{C_1} f \tilde{v}_{n_2} \vdash_{C_1} f \tilde{v}_{n_3} \vdash_{C_1} \cdots \]
This contradicts the assumption that $\text{REC}_P(f)$ is well-founded. $\square$

**Theorem 3.3** (Completeness) If $P$ is fair terminating, then for every function $f$, $\text{REC}_P(f)$ is well-founded.

**Proof** The proof is by contradiction. Suppose that $\text{REC}_P(f)$ is not well-founded for some $f$. Then there must be an infinite sequence
  \[ f \tilde{v}_1 \vdash_{C_1} f \tilde{v}_2 \vdash_{C_1} f \tilde{v}_3 \vdash_{C_1} \cdots \]
By the definition of $\vdash_{C_1}$, there exist $E_1, E_2, \ldots$ where $\text{main } \vdash_{\ast} E_1[f \tilde{v}_1]$ and $f \tilde{v}_1 \vdash_{\ast} E_{i+1}[f \tilde{v}_{i+1}]$ with $\sigma_i \models_{\text{fair}} C$ for every $i \geq 1$. Thus, we have an infinite reduction sequence $\pi$:
  \[ \text{main } \vdash_{\ast} E_1[f \tilde{v}_1] \vdash_{\ast} E_2[f \tilde{v}_2] \vdash_{\ast} \cdots \]
where for each $\sigma_i$ ($i \geq 1$), $B$ occurs if $A$ occurs for every $(A, B) \in C$. Suppose that $(A, B) \in C$ and $A$ occurs infinitely often in $\pi$. Then, there exist infinitely many $i$’s such that $A$ occurs in $\sigma_i$. But then $B$ also occurs for each of such $i$’s. Thus, $B$ also occurs infinitely often. This implies that $\pi$ is an infinite fair reduction sequence, hence a contradiction. $\square$
4. From Fair Binary Reachability to Reachability

This section describes a method for reducing a fair binary reachability problem to a plain reachability problem via program transformation; the latter problem can be solved by an off-the-shelf reachability checker for functional programs [29, 30].

4.1 Transformation

The transformation is an extension of that by [22] proposed for a reduction from (plain) binary reachability to reachability. To ease the exposition, we define the following terminology. Let \( \pi \) be a possibly infinite reduction sequence from \( \text{main} \). Let us write \( \pi(i) \) for the \( i \)th element of \( \pi \). That is,

\[
\text{main} = \pi(1) \xrightarrow{\sim} \pi(2) \xrightarrow{\sim} \pi(3) \cdots
\]

for some \( \ell_1, \ell_2, \ldots \). For total applications \( f \overline{w} \) and \( f \overline{v} \) (i.e., \( \text{arity}(f) = |\overline{w}| = |\overline{v}| \) occurring in \( \pi \), we say that \( f \overline{w} \) is an ancestor call of \( f \overline{v} \) in \( \pi \) if \( \pi(i_1) = E_1[f \overline{w}] \) and \( \pi(i_2) = E_1[E_2[f \overline{v}]] \) for some \( i_1 < i_2 \).\( E_1, E_2 \) and \( E \). We omit the reduction sequence \( \pi \) and simply say that \( f \overline{w} \) is an ancestor call of \( f \overline{v} \) when it is clear from the context. Note that \( f \overline{w} \) is an ancestor call of \( f \overline{v} \) in \( \pi \) if and only if \( f \overline{w}, f \overline{v} \in \text{Call}(\pi) \) (cf. Section 1.1). The terminology is adopted from [22], and expresses the fact that \( f \overline{w} \) occurs as an ancestor node of \( f \overline{v} \) in the call tree of \( \pi \).

As in [22], to check \( \text{REC}_{P,C}(f) \subseteq D_f \), the arguments \( \overline{w} \) of an ancestor call to \( f \) is passed around as an auxiliary argument of each function, and when \( f \) is called with a new argument \( \overline{w} \), we assert that \( D_f(\overline{w}, \overline{v}) \) holds. A further twist is required, however. We also need to keep information about the set \( \sigma \) of events that have occurred since the ancestor call of \( f \), and pass it around as another auxiliary argument. When \( f \) is called with a new argument \( \overline{w} \), we assert \( D_f(\overline{w}, \overline{v}) \) only if \( \sigma \models \overline{C} \).

We first explain the transformation informally, by using the program in Example 2.2. Let \( D_f \) be \( \{(x', x) \mid x' > x \geq 0\} \) and \( C \) be \( \{A, \text{Never}\} \). To check that \( \text{REC}_{P_{op}, C}(f) \subseteq D_f \), we transform \( P_{op} \) to the following program:

\[
\begin{align*}
\text{repeat} & \quad \text{swg} = \text{let} \quad \overline{x} = \text{rand}() \text{ in} \\
& \quad \text{let} \quad \overline{s} = g \text{ swx in repeat swg} \\
& \quad \text{if} \quad s \text{ w x} = \text{assert}(s = \text{false} \Rightarrow w > x > 0) \\
& \quad \text{let} \quad s = \overline{s} \\
& \quad \text{if} \quad \overline{rand}() \quad \text{then} \quad (\text{false, x}) \quad \text{else} \quad (s, w) \text{ in} \\
& \quad \text{if} \quad x > 0 \quad \text{then} \quad s \text{ w (x - 1)} \quad \text{else} \quad (\text{true, } \overline{x}) \\
\text{main} & \quad \text{swy} = \text{repeat swf}
\end{align*}
\]

Here, we have added two arguments \( s \) and \( w \) before each of the original function argument. The argument \( w \) holds the value of the argument of an ancestor call to \( f \), and \( s \) is a Boolean flag that expresses whether the event \( A \) has occurred since the ancestor call \( f \). The return value of each function call is now a pair consisting of the flag and the original value. At the beginning of the body of \( f \), it is asserted that if an event \( A \) has not occurred, \( w > x > 0 \) (i.e., \( D_f(w, x) \)) must hold. If the assertion fails, the program is aborted. Otherwise, the pair \( (s, w) \) is nondeterministically kept or updated to \( (\text{false}, x) \); in the latter case, the relationship between the present call and a future call to \( f \) is checked. Thanks to this non-determinism, whenever there is a call \( f \overline{w} \xrightarrow{\sim} \pi_{op} \rightarrow E[f e] \) in the original program (where the second call is not necessarily directly called from the first one), the transformed program asserts that \( w > v \geq 0 \) if \( \sigma \models \overline{C} \). Therefore, \( \text{REC}_{P_{op}, C}(f) \subseteq D_f \) if and only if the assertion in \text{main} succeeds never fails. Here, assume that \( \bot \) is a special value that satisfies \( \bot > n \) for any integer \( n \).

There are some important subtleties to note about the transformation. First, while we maintain and pass around two types of auxiliary information (i.e., arguments to \( f \) and the event information), the two are passed in different ways. Specifically, the event information is passed through the sequential flow of the program execution. This requires each function to return the event information as an auxiliary output so that the caller is able to use the information in the subsequent computation. For example, in the body of \( \text{repeat} \), the event information returned by \( g \text{ swx} \) is bound to \( s' \) and fed to the subsequent recursive call \( \text{repeat s'w} \). By contrast, the argument information is only passed from the ancestor call to its descendants. Therefore, for example, the past arguments to \( f \) passed in the two calls that happen in the body of \( \text{repeat} \) are both \( w, g \text{ swx} \) and \( \text{repeat s'w} \). This is done so that detecting the fairness of a transition sequence is done by inspecting the sequential flow of the execution sequence, while checking for disjunctive well-foundedness is done on the calling relation over such a sequence.

Secondly, the transformation passes the auxiliary information at every function application site. This is needed because it is impossible in general to statically decide which indirect function application is a total application, or which is a call to the target function (\( f \) in the example above). We note that it is possible to soundly eliminate some of the redundancy via a static analysis, and we use such an optimization in our implementation. However, it is in general impossible to completely decide a priori which function is called in what context. The strength of our approach is that the program transformation delegates such tasks to the backend reachability checker which would reason about such difficult reachability queries if they are needed to prove the goal.

We now formalize the transformation. The target language is obtained by extending the source language as follows.

\[
\begin{align*}
e & ::= \cdots | \text{fail} | \text{let} \ x_1, \ldots, x_k = e_1 \text{ in } e_2 \ \\
v & ::= \cdots | (v_1, \ldots, v_k) | \bot
\end{align*}
\]

Here, \text{fail} aborts the program, and the special value \( \bot \) is used as the initial value for the argument of an ancestor call to \( f \). We write \text{assert}(e) \text{ for } \text{let } x = e \text{ in } x \text{ if } x \text{ then } e \text{ else } \text{fail}.

The evaluation contexts and the labeled reduction relation are extended by:

\[
\begin{align*}
E & ::= \cdots | \text{let} \ x_1, \ldots, x_k = E \text{ in } e \\
E[\text{let} \ x_1, \ldots, x_k = (v_1, \ldots, v_k)] & \rightarrow \left( E[1/v_1, x_1, \ldots, v_k/x_k] : e \right) \\
E[\text{fail}] & \rightarrow \bot
\end{align*}
\]

We remark that we have defined \text{fail} to be non-terminating. This is done for the technical reason of simplifying the statement of Theorem 4.1. Since the labels for reductions are not important in the target language, we omit them below.

We define the transformation in a top-down manner. A program \( P = \{f_1 : x_1 = e_1, \ldots, f_n : x_n = e_n\} \) is transformed to:

\[
[P]_{f,D,C} = \bigcup_{i \in \{1, \ldots, n\}} [f_i : x_i = e_i]_{f,D,C}.
\]

Here, each function definition \( g \ x_1 \cdots x_k = e \) is translated to the set \( [g \ x_1 \cdots x_k = e]_{f,D,C} \) of function definitions, given by:

\[
\begin{align*}
[g \ x_1 \cdots x_k = e]_{f,D,C} & = \{g \swx : x_1 = (s, g^{(1)}(x_1)), \ldots, g^{(k)}(x_1) : x_2 = (s, g^{(2)}(x_1, x_2)), \ldots, g^{(k)}(x_1, \ldots, x_{k-1}) : x_k = e'\}
\end{align*}
\]

As in the transformation of \( P_{op} \) explained above, we add two auxiliary arguments \( s \) and \( w \) before each argument. As in the example, \( w \) is the value of the argument of an ancestor call to \( f \) (which is actually a tuple, since \( f \) may take more than one argument), and \( s \),
called an event history, keeps information about the events that have
occurred since the call of $w$. If $C = \{(A_1, B_1), \ldots , (A_n, B_n)\}$,
then $s$ is a nested tuple of the form:

$$(p_1, q_1), \ldots , (p_n, q_n))$$

where $p_i$ (resp.) is a Boolean that expresses whether $A_i$ (resp.) has occurred.

Since each partial application of $g$ should return a pair consisting
of an event history and a closure, we have prepared auxiliary
functions $g^{(1)}, \ldots , g^{(k-1)}$. The body $e'$ of $g^{(k-1)}$ is:

assert(fair, ⇒ $D^\#(w_{k-1}, (x_1, \ldots , x_k));$

let $(s, w) =$

if rand() then $(s_{\text{init}}, (x_1, \ldots , x_k))$ else $(s, w)$ in

$[e]_{s,w,C}$

if $g = f$ and $[e]_{s,w,C}$ otherwise (where the transformation
$[e]_{s,w,C}$ for expressions is given later). Here, $s_{\text{init}}$ denotes the expression

let $((p_1, q_1), \ldots , (p_n, q_n)) =$ s in

$$(p_1 \Rightarrow q_1) \& \cdots \& (p_n \Rightarrow q_n),$$

and $s_{\text{init}}$ denotes the value

$$(false, false, \ldots , false, false).$$

$D^\#(w, v)$ is an expression such that $D^\#(w, v)$ evaluates to true
if $v = 1$ and $[v]_{s,w,C}$ with $((w_1, \ldots , w_k), (v_1, \ldots , v_k)) \in D$ or (ii) $v = \perp$, and evaluates
to false otherwise. In the definition of $e'$ above, $s$ contains
information about the sequence $\sigma$ of events that have occurred since
an ancestor call to $f$, and $fair$, expresses whether $\sigma$ $\Rightarrow_{\text{fin}} C$ holds.

The transformations of expressions and values, denoted by
$[\cdot]_{s,w,C}$ and $[\cdot]_{\perp}$ respectively, are defined as shown in Figure 5.
Note that, after the transformation, an expression returns a pair consisting
of an updated event history and the value. On the third line,
s$\{A := \text{true}\}$ is an expression for updating the event
history by replacing the elements corresponding to $A$ with $true$.
More precisely, if $C = \{(A_1, B_1), \ldots , (A_n, B_n)\}$, then

$s\{A := \text{true}\}$ is:

let $((p_{1,1}, p_{2,1}), \ldots , (p_{1,n}, p_{2,n})) =$ s in

$$(p'_{1,1}, p'_{2,1}), \ldots , (p'_{1,n}, p'_{2,n})$$

where $p'_{i,j}$ is true if $A_i = A$ and $p_{i,j}$ otherwise.

In the following, we fix the fairness constraint and omit it
in the transformation notation (e.g., we write $[P]_{f,D,C}$ instead of
$[P]_{f,D,C}$). The following theorems guarantee that our reduction

Therefore, if $v$ contains a closure, the target language needs to have an
expressive power to inspect the contents of the closure; we can realize it by
choosing an appropriate representation of a closure.

from fair binary reachability to plain reachability is sound and


to complete.

Theorem 4.1 (Correctness of the transformation)

Suppose $[P]_{f,D} = P'$. Then, $\text{REC}_{P,C}(f) \not\subseteq D$ if and only if

main $\not\Rightarrow_{\perp} \Rightarrow_{P'} \text{fail}$.

The rest of this section is devoted to the proof of the theorem
above, which is divided into the following two sub-theorems:

Theorem 4.2 (Soundness of the transformation)

Suppose $[P]_{f,D} = P'$ and $\text{REC}_{P,C}(f) \not\subseteq D$. Then, if $\text{REC}_{P,C}(f) \not\subseteq D$

fail then $\text{main} \not\Rightarrow_{\perp} \Rightarrow_{P'} \text{fail}$.

Theorem 4.3 (Completeness of the transformation)

Suppose $[P]_{f,D} = P'$. Then, if $\text{main} \not\Rightarrow_{\perp} \Rightarrow_{P'} \text{fail}$ then

$\text{REC}_{P,C}(f) \not\subseteq D$.

4.2 Proof of Soundness (Theorem 4.2)

The soundness follows from the following properties: (i) each reductive
of an expression $e$ in the source program is simulated by
the reductions of the corresponding expression $[e]_{s,w}$ of the transformed
program (Lemma 4.6), and (ii) the argument of an ancestor
call of $f$ is correctly recorded in the transformed program
(Lemma 4.7).

We first prepare some definitions and auxiliary lemmas. We
extend the transformation to contexts by:

\begin{align*}
\Gamma & \Rightarrow [e]_{s,w} = \Gamma \Rightarrow (s, w) \in [e]_{s,w,C} \\
\Gamma & \Rightarrow \ell \Rightarrow [e]_{s,w} = \Gamma \Rightarrow \ell \Rightarrow (s, w) \in [e]_{s,w,C} \\
\Gamma & \Rightarrow (\ell_1, \ell_2) \Rightarrow [e]_{s,w} = \Gamma \Rightarrow (\ell_1, \ell_2) \Rightarrow (s, w) \in [e]_{s,w,C} \\
\Gamma & \Rightarrow \ell \Rightarrow [e/x]_{s,w} = \Gamma \Rightarrow \ell \Rightarrow (s, w) \in [e/x]_{s,w,C} \\
\end{align*}

\begin{align*}
\text{Lemma 4.4} & \quad \text{If } E \text{ is a context for the source language, then } [E]_{s,w} = [E]_{s,w} \\
\text{Lemma 4.5} & \quad [v/x]_{s,w} = [v/x]_{s,w}.
\end{align*}

\begin{align*}
\text{Proof} & \quad \text{This follows by a straightforward induction on the structure of } E. \\
\text{Proof} & \quad \text{This follows by a straightforward induction on the structure of } e.
\end{align*}

Let $\sigma$ be a finite sequence of events and $s$ be an event history.
We define $\sigma(s)$ by:

$\epsilon(s) = s \quad A \cdot \sigma(s) = \sigma([s\{A := \text{true}\}])$

where $[s\{A := \text{true}\}]$ is the value of the expression $s\{A := \text{true}\}$. We write $D_0$ for the “trivial” relation such that $(\overline{v}, \overline{w}) \in D_0$
for all $\overline{v}$ and $\overline{w}$. The following is the main lemma, which states that
reductions of a source program can be simulated by those of the
transformed program.

\begin{align*}
\text{Lemma 4.6} & \quad \text{Let } P' \text{ be } [P]_{f,D_0}. \text{ If } e_1 \Rightarrow_{P} e_2, \text{ then } [e_1]_{s,w} \Rightarrow_{P'} [e_2]_{\ell(s),w}. \\
\text{Proof} & \quad \text{The proof proceeds by a case analysis on the rule used for}
\end{align*}

\begin{itemize}
  \item Case E-EV: In this case, $e_1 = E[\text{event } A]$ and $e_2 = E[s]$ with
  \ell = A. By Lemma 4.4 and the definition of the transformation, we have:
  \begin{align*}
  [e_1]_{s,w} & = [E]_{s,w}([s\{A := \text{true}\}, s]) \\
  [e_2]_{\ell(s),w} & = [E]_{s,w}([\ell(s), s])
  \end{align*}
  \text{Thus, we have } [e_1]_{s,w} \Rightarrow_{P'} [e_2]_{\ell(s),w} \text{ as required.}
\end{itemize}
• Case E-Clos: In this case, $e_1 = E[(g v_1 \ldots v_{i-1})v_i]$ and $e_2 = E[(g v_1 \ldots v_i)]$ with $\ell = \epsilon$ and $i < \text{arity}(g)$. By Lemma 4.4 and the definition of the transformation, we have:

\[
\begin{align*}
[e_1]_{s,w} &= \left[ E \right]_{w}[(g^{(i-1)})[v_1] \cdot \ldots \cdot [v_{i-1}] v_i] s w [v_i] \\
[e_2]_{(*)} &= \left[ E \right]_{w}[(\ell s, (g^{(i)})[v_1] \cdot \ldots \cdot [v_i])] \\
\end{align*}
\]

By inspection of the reduction rules, we have:

\[
\begin{align*}
[e_1]_{s,w} &\rightarrow^* \left[ E \right]_{w}[(g^{(i-1)})[v_1] \cdot \ldots \cdot [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[(v_1) \cdot \ldots \cdot [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\end{align*}
\]

Since $\ell = s$, we have $[e_1]_{s,w} \rightarrow^* [e_2]_{(*)}$ as required.

• Case E-Call: In this case, $e_1 = E[(g v_1 \ldots v_{i-1}) v_i]$ and $e_2 = E[[v_1/x_1], \ldots, [v_i/x_i]] e$ with $g x_1 \ldots x_i = e \in P$ and $\ell = \epsilon$. If $g \neq f$, then we have:

\[
\begin{align*}
[e_1]_{s,w} &= \left[ E \right]_{w}[(g^{(i-1)})[v_1] \cdot \ldots \cdot [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\end{align*}
\]

By Lemma 4.5 and the definition of the transformation. If $g = f$, then we have:

\[
\begin{align*}
[e_1]_{s,w} &= \left[ E \right]_{w}[(g^{(i-1)})[v_1] \cdot \ldots \cdot [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\rightarrow^* \left[ E \right]_{w}[[v_1] v_1, \ldots, [v_{i-1}] v_i] s w [v_i] \\
\end{align*}
\]

as required.

The following lemma guarantees that whenever there is a call $f v_1 \ldots v_k$ in a source program, the argument $[v_1, \ldots, v_k]$ is correctly recorded in some reduction sequence of the transformed program.

\textbf{Lemma 4.7} Suppose $f x_1 \ldots x_k = e \in P$ and $[P]_{f,D_0} = P'$. Then,

\[
(f v_1 \ldots v_{k-1} v_k) s,w \rightarrow^* P' [(v_1/x_1, \ldots, v_k/x_k)] e_{\text{start}, w'}
\]

with $w' = ([v_1], \ldots, [v_k])$.

\textbf{Proof} This follows immediately from the definition of the transformation and Lemma 4.5.

\textbf{Proof of Theorem 4.2} Let $[P]_{f,D} = P'$ and $f w_1 \ldots w_k \rightarrow^* P$.

\[
(f v_1 \ldots w_k) s,w \rightarrow^* P' [(v_1/x_1, \ldots, w_k/x_k)] e_{\text{start}, w'}
\]

with $w' = ([v_1], \ldots, [w_k])$.

\textbf{Proof} This follows immediately from the definition of the transformation and Lemma 4.5.

We have either main $s_{\text{init}} \downarrow * \rightarrow^* P'$, or main $s_{\text{init}} \downarrow * \rightarrow^* P'$, or $E_s[[f v_1 \ldots v_{k-1}] v_k]_{\text{start}, (w'_1, \ldots, w'_{k-1})}$.[[3]]

Thus, we have either $\text{main } s_{\text{init}} \downarrow * \rightarrow^* P'$, or $\text{main } s_{\text{init}} \downarrow * \rightarrow^* P'$, or $E_s[[f v_1 \ldots v_{k-1}] v_k]_{\text{start}, (w'_1, \ldots, w'_{k-1})}$.

Note that the only difference between $P''$ and $P'$ is that $P'$ has stronger assertion conditions; thus, reductions under $P'$ may have only more possibilities to fail. In the latter case, we have:

\[
(f v_1 \ldots v_{k-1} v_k) s,w \rightarrow D'' (w', v')
\]

Here, $w' = (w'_1, \ldots, w'_{k-1})$ and $v' = (v'_1, \ldots, v'_{k-1})$ with $v'_i = [v_i] v_i$.

Note that $\text{fair}_{s,w} \rightarrow D'' (w', v')$ evaluates to true by the condition $\sigma_1, C$, and $D'' (w', v')$ evaluates to false by the assumption $(w_1, \ldots, w_k, v_1, \ldots, v_k) \notin D$. $\square$

4.3 Proof of Completeness (Theorem 4.4)

The completeness follows from the property that reductions of the transformed program can be simulated by those of the source program, as stated in Lemma 4.8 below. The statement is a little more complex than the converse (Lemma 4.6), due to the non-determinism in auxiliary reduction steps of the transformed program.

We call an expression $e$ an instruction if $e$ matches the left-hand side of a reduction rule in Section 2 with $E = []$.

\textbf{Lemma 4.8} Suppose $[P]_{f,D} = P'$. If $e_1$ is an instruction and $e_2$ is $\hat{v}$, then there exist $e_2$ and $\ell$ such that $e_1 \rightarrow^* e_2$ and $e_2 \rightarrow^* e_3$ is obtained without using the rule for function calls, or (ii) $e_1 \rightarrow^* e_2$ with $\hat{v}$ or $\hat{\ell}$ and $\sigma_2 = \text{start}$. Furthermore, either (i) $s = \ell$ and $w' = w$, or (iv) $s = \text{start}$ and $w' = ([v_1], \ldots, [v_k])$ with $e_1 = (f v_1 \ldots v_{k-1}) v_k$.

\textbf{Proof} The proof proceeds by induction on the length of the reduction sequence $[e_1]_{s,w} \rightarrow^* e_3$, with a case analysis on the shape of $e_1$. We discuss only the main cases; the other cases are straightforward.

• Case $e_1 = \text{event } A$: In this case, we have

\[
[e_1]_{s,w} = (s(A := \text{true}), *)
\]

Thus, the result holds for $e_2 = \ast$ and $\ell = A$.

• Case $e_1 = (g v_1 \ldots v_{i-1}) v_i$: In this case, we have

\[
[e_1]_{s,w} = (g^{(i-1)})[v_1] \cdot \ldots \cdot [v_{i-1}] v_i s w [v_i] v_i.
\]

If $i < \text{arity}(g)$, then the result holds for $e_2 = (g v_1 \ldots v_i)$ and $\ell = \epsilon$.

If $i = \text{arity}(g)$ and $g \neq f$ with $g x_1 \ldots x_i = e \in P$, then the result holds for $e_2 = [v_1/x_1, \ldots, v_i/x_i] e$ and $\ell = \epsilon$.

If $i = \text{arity}(g)$ and $g = f$ with $g x_1 \ldots x_i = e \in P$, then the sequence $[e_1]_{s,w} \rightarrow^* e'$ must be a prefix or postfix of the reduction sequence $[e_1]_{s,w} \rightarrow^* [v_1/x_1, \ldots, v_i/x_i] e_{\text{start}, (v_1, \ldots, v_i)}$ or $[e_1]_{s,w} \rightarrow^* [v_1/x_1, \ldots, v_i/x_i] e_{\text{start}, (v_1, \ldots, v_i)} e_{\text{start}, (v_1, \ldots, v_i)}$.

Thus, the result holds for $e_2 = [v_1/x_1, \ldots, v_i/x_i] e$ and $\ell = \epsilon$.

\square

The following lemma guarantees that if $w$ is recorded in the target program as the argument of an ancestor call for $f$, $f$ has indeed been called with $w$ before.

\textbf{Lemma 4.9} Suppose $[P]_{f,D} = P'$. If $[e_1]_{s,w} \rightarrow^* E_s[e_2]_{s',w'}$ with $w \neq w'$ and $e_2$ is a function application, then
Proof

This follows by induction on the length of the reduction sequence \([e_1]_{s,w} \rightarrow^* \rho \ E_1[\langle f_1 \cdots v_{k-1} v_k \rangle_{\varepsilon',w'}]\) and
\([f_1 \cdots v_{k-1} v_k \rangle_{\varepsilon',w'} \rightarrow^* \rho \ E_2[\langle e_2 \rangle_{\varepsilon',w'}]\)
with \(E = E_1[E_2]\) and \(w' = (v_1, v_2, \ldots, v_h)\).

\[\begin{align*}
\text{Proof of Theorem 4.3} & \\
& \text{Suppose it must be the case that:} \\
& \begin{cases}
\text{with the expression:} \\
\text{an instruction. By the assumption} \\
\text{of Lemma 4.8 does not hold, since} \\
\text{must be reducible. So,} e_1 \text{can be decomposed to} E_3[e_1' \text{ where} e_1' \text{is an instruction. By the assumption} \\
\text{have one of the following cases.} \\
\text{for functional programs [13, 20, 23]. We translate the} \\
\text{rankings function inference. As described in Section 1.1} \\
\text{backend reachability checker, and Z3 [8] as a constraint solver for} \\
\text{functional programs to OCaml, and we translate the} \\
\text{termination (for ones from [20, 23] where the properties are given as temporal} \\
\text{logic formulas or automaton inclusion problems). We have} \\
\text{conducted the experiments on a machine with Intel Core i7-3930K} \\
\text{inference for functional programs [13, 20, 23]. We translate the} \\
\text{Web interface of the implementation and the benchmarks used in} \\
\text{the experiments are available on the web.}\end{cases}
\end{align*}\]

Table 1. The experiment results.

<table>
<thead>
<tr>
<th>program</th>
<th>cycle1</th>
<th>cycle2</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro</td>
<td>3</td>
<td>14</td>
<td>11.492</td>
</tr>
<tr>
<td>repeat</td>
<td>4</td>
<td>12</td>
<td>2.276</td>
</tr>
<tr>
<td>closure</td>
<td>6</td>
<td>18</td>
<td>9.76</td>
</tr>
<tr>
<td>hofmann-1</td>
<td>13, 14</td>
<td>2</td>
<td>0.232</td>
</tr>
<tr>
<td>hofmann-2</td>
<td>13, 14</td>
<td>3</td>
<td>1.032</td>
</tr>
<tr>
<td>koskinen-1</td>
<td>20</td>
<td>7</td>
<td>43.344</td>
</tr>
<tr>
<td>koskinen-2</td>
<td>20</td>
<td>5</td>
<td>3.412</td>
</tr>
<tr>
<td>koskinen-3-1</td>
<td>20</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>koskinen-3-2</td>
<td>20</td>
<td>4</td>
<td>2.216</td>
</tr>
<tr>
<td>koskinen-3-3</td>
<td>20</td>
<td>6</td>
<td>4.964</td>
</tr>
<tr>
<td>koskinen-4</td>
<td>20</td>
<td>10</td>
<td>132.552</td>
</tr>
<tr>
<td>lester [23]</td>
<td>8</td>
<td>36</td>
<td>38.356</td>
</tr>
</tbody>
</table>

Notes that only place where the translated program may fail is the expression:

\[\text{assert(fair} \Rightarrow \text{D}^\#(w_k, (x_1, \ldots, x_k))\).

By induction on the length of the reduction sequence and Lemma 4.8, it must be the case that:

\[\begin{align*}
\text{main $s_{\text{init}} \perp \ast \rightarrow \rho \ E_3[v]_{s,w} \rightarrow^*} & \\
\text{E_4[\langle e_2 \rangle_{\varepsilon',w'}]} & \\
\text{fair $s' \rightarrow \text{true}} & \\
\text{D}^\#((w_1, \ldots, w_k), v_1, \ldots, v_h) & \rightarrow \ast \text{false}
\end{align*}\]

Thus, we have the required result.

5. Implementation and Experiments

We have implemented a prototype of our fair-termination verification method for a subset of OCaml. We use MoCHi [19] as the backend reachability checker, and Z3 [8] as a constraint solver for ranking function inference. As described in Section 1.1 Termination Argument Inference (and [22]), we use the implicit parameter technique to obtain higher-order termination arguments.

We have tested our tool on examples from this paper, and benchmark programs taken from previous work on temporal property verification for functional programs [13, 20, 23]. We translate the

\[\text{const}\ x\ y = x \text{; finish} \ x = \text{event A; finish} \ x \text{; if} \ n > 0 \text{then} \begin{cases}
\text{f} \ g = \text{let} \ n = g \ast \text{in} \\
\text{if} \ n > 0 \text{then} \begin{cases}
\text{main} \ x = \text{let} \ n = \text{rand}() \text{in} \begin{cases}
\end{cases}
\end{cases}
\end{cases} \]

The benchmark is an example where a higher-order termination argument (i.e., disjunctive well-founded relation over function values) is required for verification. It is interesting to note that the other benchmarks, including the ones from the previous literature, can be verified with only first-order termination argument (but higher-order reasoning is required at the rest of the verification pro-

6 http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/fair_termination/
The benchmarks hofmann-1 and hofmann-2 are from a recent work by Hofmann and Chen [13]. Specifically, hofmann-1 (resp. hofmann-2) is the first (resp. second) example from Appendix A of the extended technical report [14] (hofmann-2 also appears as an example in Section 5.4 of [13]). Their paper shows how these benchmarks can be verified manually in their framework, but it does not show how to automate the process.

The benchmarks koskinen-xxx’s are from a recent paper by Koskinen and Terauchi [20]. They are taken from Figure 10 of the paper where koskinen-1 is REDUCE, koskinen-2 is RUMBLE, koskinen-3-x’s are RUMBLE, and koskinen-4 is ALTERNATE INEVITABILITY. Their paper presents the temporal properties to be verified as temporal logic formulas, and we have translated them to fair termination for our experiments in the manner described in Section 2. We note that RUMBLE is translated into (the conjunction of) three sub-problems: koskinen-3-1, koskinen-3-2 and koskinen-3-3. During the experiments, we have actually discovered a bug in ALTERNATE INEVITABILITY in their paper, and the result presented here is for a version with the bug corrected. As with the work by Hofmann and Chen, their paper shows how these benchmarks can be verified manually in their framework, but it does not show how to automate the process.

The benchmark lester is the example from Appendix H.1 of the paper by Lester et al. [23]. Their verification method can only handle finite data, simply-typed higher-order functional programs, so they used Church numerals to represent natural numbers, and manually translated a loop over integers into an iteration over Church numerals. The version we verify in the experiments directly uses integers. Also, their paper presents the temporal property to be verified as a temporal logic formula and an automaton inclusion problem, and we have translated it to fair termination for our experiments in the manner described in Section 2.

As seen in Table 1, the results show that our implementation successfully verifies all of the benchmarks. The implementation is able to verify the benchmarks quite quickly, except for intro, koskinen-1, koskinen-4, and lester. The performance bottleneck appears to be the backend reachability checker MoCHI [19] which is taking a rather long time to verify the reachability problems generated in the verification of these benchmarks. In particular, the benchmarks koskinen-1, koskinen-4, and lester, after the CPS/HORS (higher-order recursion scheme) translation done within MoCHI, produce large reachability verification instances. For intro, the HORS model checker within MoCHI seems to be the main bottleneck, and we leave for future work to analyze why the HORS model checker underperforms on the recursion schemes generated in this benchmark. We remark that this is not a fundamental limitation with our fair-termination verification approach, and we expect further advances in reachability verification to allow our approach to verify instances like these more quickly.

6. Conclusion

We have presented an automatic approach to temporal property verification of higher-order functional programs. Previous automated approaches to this class of programs could only handle finite data programs or only safety properties or plain termination, and our work is the first to be able to verify arbitrary ω-regular properties of infinite data high-order functional programs. Our approach combines and extends the techniques from the recent work on binary-reachability based approaches to plain termination verification of higher-order functional programs [22] and fair termination verification of imperative programs [2], and reduces the temporal property verification problem soundly and completely to fair binary reachability verification problems over calling relations. A key contribution of our approach includes the novel program transformation that correctly tracks the calling relation and the event occurrence information through the higher-order control flow.

Acknowledgments

We thank the anonymous reviewers for useful comments. This work was partially supported by MEXT Kakenhi 26330082, 25280023, 15H05706, 23220001, 25730035, and 25280020, and JSPS Core-to-Core Program (A. Advanced Research Networks).

References


