Automatically Disproving Fair Termination of Higher-Order Functional Programs

Keiichi Watanabe Ryosuke Sato Takeshi Tsukada Naoki Kobayashi

The University of Tokyo, Japan

{watanabe,ryosuke,tsukada,koba}@kb.is.s.u-tokyo.ac.jp

Abstract

We propose an automated method for *disproving* fair termination of higher-order functional programs, which is complementary to Murase et al.'s recent method for *proving* fair termination. A program is said to be *fair terminating* if it has no infinite execution trace that satisfies a given fairness constraint. Fair termination is an important property because program verification problems for arbitrary ω-regular temporal properties can be transformed to those of fair termination. Our method reduces the problem of disproving fair termination to higher-order model checking by using predicate abstraction and CEGAR. Given a program, we convert it to an abstract program that generates an approximation of the (possibly infinite) execution traces of the original program, so that the original program has a fair infinite execution trace if the tree generated by the abstract program satisfies a certain property. The method is a non-trivial extension of Kuwahara et al.'s method for disproving plain termination. We implemented a prototype verification tool based on our method and confirmed its effectiveness.

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—Mechanical verification

General Terms Languages, Verification

Keywords Automatic Verification, Fair Termination, Higher-Order Programs, Predicate Abstraction, Higher-Order Model Checking, CEGAR

1. Introduction

There has recently been rapid progress in automated (or semi-automated) techniques for verification of higher-order functional programs [11, 15–17, 19, 21, 22, 26, 27, 29, 30, 33, 35, 36]. Verification methods have been proposed for various properties, including safety properties [19, 33], termination [11, 21], non-termination [22], and fair termination [24].

In the present paper, we propose a method for *disproving* fair termination, which plays a role complementary to Murase et al.'s method [24] for proving fair termination. Fair termination is a key to proving temporal properties of programs: Vardi [34] has shown that the verification of any ω-regular temporal property can be reduced to that of fair termination. Cook et al. [4] applied the reduction to verification of liveness properties of imperative programs. Murase et al. [24] has recently proposed a method for *proving* fair termination of higher-order programs, but their method cannot be used for *disproving* it.

Let us briefly explain the notion of fair termination. A (possibly infinite) execution trace σ of a program is *fair* with respect to a (Streett) fairness constraint $\{(A_1, B_1), \dots, (A_n, B_n)\}$ (where A_i, B_i are called *events*) when, for every i, if the event A_i occurs infinitely often in σ, so does the event B_i. For example, an infinite trace $(AB)^\omega = ABABAB\cdots$ is fair with respect to the constraint $\{(A, B)\}$, but $A^\omega = AAA\cdots$ is not. A program P is *fair terminating* when P has no fair infinite execution traces. Many temporal properties can be naturally reduced to fair termination. For example, let `Never` be an event that never happens. Then, the fair termination of a program with respect to the constraint $\{(A, \texttt{Never})\}$ means that A occurs infinitely often in any infinite run of the program; notice that an infinite run that contains only finitely many A's is fair, violating the fair termination condition. For another example, to check that a function f is eventually called in any infinite run of the program, it suffices to replace the function f with a function to raise event A infinitely often, and check that the resulting program is fair terminating with respect to $\{(A, \texttt{Never})\}$. See [34] for a general reduction from ω-regular property verification to fair termination verification.

As a concrete example for illustrating why fair termination is of interest, let us consider the following program P_0, consisting of mutually recursive function definitions:

$$\begin{aligned} \mathit{rand}\ () &= *_{\text{int}} \\ \mathit{randnneg}\ () &= \texttt{let}\ x = \mathit{rand}\ ()\ \texttt{in} \\ &\quad\ \texttt{if}\ 0 \le x\ \texttt{then}\ x\ \texttt{else}\ \mathit{randnneg}\ () \\ \textbf{main} &= \mathit{randnneg}\ (). \end{aligned}$$

Here, $*_{\text{int}}$ is a special expression that generates a random integer; thus the function rand returns a random integer. The function $\mathit{randnneg}$ returns a non-negative random integer, by repeatedly calling rand and then returning the result only if it is non-negative.

Suppose that we wish to prove that the program above is terminating; note that the program is terminating indeed, since rand must eventually return a non-negative integer by the assumption on randomness. Ordinary termination verification methods [11, 21] would, however, fail to prove so, since they would treat $*_{\text{int}}$ as a non-deterministic function to return an integer. Using fair termination, we can incorporate the assumption that rand eventually returns a non-negative integer. Let us instrument the program above,

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...
http://dx.doi.org/10.1145/2951913.2951919

as the following program P_1:

$$\begin{aligned} rand\ () &= \texttt{let}\ r = *_{\text{int}}\ \texttt{in} \\ &\quad \texttt{if}\ 0 \le r\ \texttt{then}\ (\textbf{event}\ B;\ r)\ \texttt{else}\ (\textbf{event}\ A;\ r) \\ randnneg\ () &= \texttt{let}\ x = rand\ ()\ \texttt{in} \\ &\quad \texttt{if}\ 0 \le x\ \texttt{then}\ x\ \texttt{else}\ randnneg\ () \\ \textbf{main} &= randnneg\ (). \end{aligned}$$

Here, we have inserted event expressions (**event** B and **event** A), to signal whether the value returned by $rand$ is non-negative or not. Then, the fairness assumption about $rand$ can be expressed by $\{(A, B)\}$,[1] and fair termination of the program under $\{(A, B)\}$ means that the program terminates if $rand$ is fair. The program P_1 above is indeed fair terminating under $\{(A, B)\}$, and can be proved so by the method of Murase et al. [24].

Now we consider the following program P_2, which is a variation of P_1:

$$\begin{aligned} rand\ () &= \texttt{let}\ r = *_{\text{int}}\ \texttt{in} \\ &\quad \texttt{if}\ 0 \le r\ \texttt{then}\ (\textbf{event}\ B;\ r)\ \texttt{else}\ (\textbf{event}\ A;\ r) \\ randpos\ () &= \texttt{let}\ x = rand\ ()\ \texttt{in} \\ &\quad \texttt{if}\ 0 < x\ \texttt{then}\ x\ \texttt{else}\ randpos\ () \\ \textbf{main} &= randpos\ (). \end{aligned}$$

Notice that the branching condition in the second function has been changed to $0 < x$. The program is not fair terminating under $\{(A, B)\}$. If $rand$ returns 0 and a negative integer alternately, the event sequence is $BABABA\cdots$, which is fair, but the program is not terminating.

As already mentioned, the goal of the present paper is to automatically disprove fair termination (i.e., to prove that a given program has a fair infinite execution trace). For the examples above, we wish to show that P_2 is *not* fair terminating under $\{(A, B)\}$. In general, disproving a liveness property is much harder than disproving a safety property. In the case of safety, one just needs to find a *finite* execution trace that violates the safety property. In contrast, in the case of liveness, we need to find an *infinite* execution trace that violates the property. Even for plain termination, it is only recent that automated methods for disproving termination [11, 22] have been obtained for higher-order functional programs.

Following Kuwahara et al.'s method for disproving *plain* termination [22], we combine higher-order model checking [28] with predicate abstraction and counterexample-guided abstraction refinement (CEGAR) to disprove fair termination. Higher-order model checking decides whether the (possibly infinite) tree generated by a given higher-order tree grammar (called a higher-order recursion scheme; HORS for short) satisfies a given regular property. For *finite data* higher-order programs where only finite base types (such as Booleans, not infinite data domains like integers and lists) are allowed, arbitrary regular properties can be decided by first transforming a program to a HORS, and then applying higher-order model checking [17, 23]. For *infinite data* programs (that may use data from infinite data domains, like integers, lists, and trees), however, we need to combine higher-order model checking with predicate abstraction.

An overall flow of our method is shown in Figure 1. In Step 1, given a higher-order functional program P (that may use infinite data domains) and a fairness constraint $\mathcal{C}$ as input, we apply predicate abstraction to generate a pair consisting of (i) an abstract program, which is a *Boolean*, higher-order, *tree-generating* functional program Q_P, and (ii) a tree automaton $\mathcal{A}_\mathcal{C}$, which describes the

[1] This actually defines a stronger assumption than needed here, that if $rand$ returns negative integers infinitely often, then it also returns non-negative integers *infinitely often*, not just once.

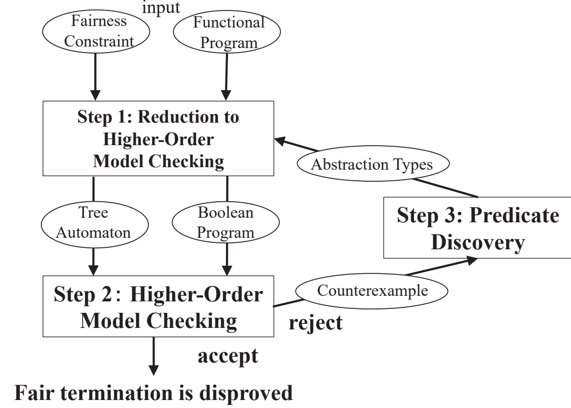

Figure 1: Overview of method.

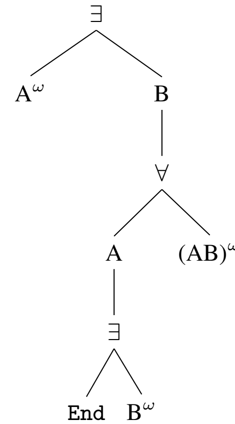

Figure 2: An example of the tree generated by an abstract program.

property that should be satisfied by the tree generated by Q_P. The tree generated by Q_P describes an abstraction of possible execution traces of the original program P. Figure 2 shows an example of the tree generated by an abstract program. The tree consists of event nodes A and B, and special nodes $\exists, \forall$, and End. In the figure, A^ω abbreviates an infinite tree path where A occurs consecutively, and $(AB)^\omega$ abbreviates an infinite path where A and B occur alternately. A $\exists$-node means that for every branch of the node, there is a corresponding execution of the original program, while a $\forall$-node means that for at least one of the branches, there is a corresponding execution of the original program (the other branches may be spurious, introduced by abstraction). An End-node describes the termination. Thus, to prove that the original program has a fair infinite execution trace (which is the goal of our method), it suffices to show, roughly speaking, (i) for each $\exists$-node, there exists a fair infinite path, and (ii) for each $\forall$-node, every branch has a fair infinite path. These conditions are formally described by the tree automaton $\mathcal{A}_\mathcal{C}$. The tree in Figure 2 indeed satisfies the property mentioned above; if we choose the right branches for both $\exists$-nodes, then we get two infinite paths $B(AB)^\omega$ and BAB^ω, which are both fair under $\{(A, B)\}$.

To see how an abstract program may be constructed, consider the following program P_3, which is a continuation passing style

(CPS) version of program P_2 above:

$$
\begin{aligned}
rand\ \ k &= f\ *_{\text{int}}\ k \\
f\ r\ k &= \texttt{if}\ 0 \le r\ \texttt{then}\ (\textbf{event}\ B;\ k\ r) \\
&\quad\ \texttt{else}\ (\textbf{event}\ A;\ k\ r) \\
randpos\ \ k &= rand\ (\lambda x.\texttt{if}\ 0 < x\ \texttt{then}\ k\,x\ \texttt{else}\ randpos\ \ k) \\
\textbf{main} &= randpos\ \lambda x.x.
\end{aligned}
$$

Suppose that we have chosen the predicate $0 < n$ for abstracting every integer n. Then we get the following abstract program Q_{P_3}:

$$
\begin{aligned}
rand\ \ k &= \exists(f\ \texttt{false}\ k, f\ \texttt{true}\ k) \\
f\ b_{0<r}\ k &= \texttt{if}\ b_{0<r}\ \texttt{then}\ B(k(b_{0<r})) \\
&\quad\ \texttt{else}\ \forall(A(k(b_{0<r})), B(k(b_{0<r}))) \\
randpos\ \ k &= rand\ (\lambda b_{0<x}. \\
&\quad\ \texttt{if}\ b_{0<x}\ \texttt{then}\ k\,b_{0<x}\ \texttt{else}\ randpos\ \ k) \\
\textbf{main} &= randpos\ (\lambda b_{0<x}.\texttt{End}).
\end{aligned}
$$

Here, the integer variables r and x have been replaced by Boolean variables $b_{0<r}$ and $b_{0<x}$, which respectively represent whether the values of r and x in the original program are positive or not. Event expressions have been replaced by unary tree constructors A and B. In addition, we have inserted tree constructors $\exists$ and $\forall$. In the body of $rand$ in P_3, the value returned by $*_{\text{int}}$ may be positive or not. Thus, in Q_{P_3}, we create two subtrees by f `false` k and f `true` k, which describe executions for the cases where negative and non-negative values have been returned respectively; we then combine them with $\exists$, to indicate that, to prove that the program has a fair infinite execution, it suffices to show that one of the subtrees has a fair infinite path. In the body of f in Q_{P_3}, the conditional branch on r in the original program has been replaced by that on $b_{0<r}$. If $b_{0<r}$ is `true`, then the original condition $0 \le r$ is also `true`, so we create a tree describing the computation of the then-branch of the original program. If $b_{0<r}$ is `false`, then the original condition $0 \le r$ may be `true` or `false`; thus we create two subtrees for describing both cases, and combine them with a $\forall$-node, to indicate that, to prove that the program has a fair infinite execution, we need to show that *both subtrees* have a fair infinite path (since, due to the abstraction, we do not know which subtree describes the actual computation of the original program). The tree generated by the abstract program is shown in Figure 3, where the two $\exists$-subtrees at the bottom are identical to the whole tree; so it is the regular tree X defined by:[2]

$$X = \exists(\forall(A(X), B(X)), B(\texttt{End})).$$

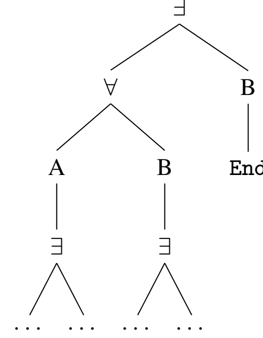

Figure 3: The tree generated by Q_{P_3}.

In Step 2, we use higher-order model checking to check whether the tree generated by the abstract program Q_P is accepted by the tree automaton $\mathcal{A}_C$; recall that $\mathcal{A}_C$ describes a sufficient condition for the original program P to have a fair non-terminating execution trace. The tree generated by the abstract program Q_{P_3} given above actually does not satisfy the sufficient condition. To show the existence of a fair non-terminating execution, for each $\exists$-node, we have to choose the left branch; however, if we choose the left branch of every $\forall$-node, we get an event sequence $A^\omega = AAA\cdots$, which is unfair with respect to $\{(A, B)\}$. In such a case, we let a higher-order model checker output a counterexample, which is a subtree of the tree generated by Q_P, showing why the tree is not accepted by $\mathcal{A}_C$. Actually, since such a subtree is infinite in general, we generate a (higher-order) tree grammar that generates the subtree as a counterexample, and pass it to Step 3. For the example above, the counterexample can be described by the grammar:

$$X = \exists(\forall(A(X), _), B(\texttt{End})).$$

In Step 3, we analyze the counterexample, and if it is spurious, discover predicates to refine the abstraction. This may sound standard, but it is actually non-trivial, since the counterexample is an *infinite* tree and it has two kinds of branching nodes $\forall$ and $\exists$.

For the example above, suppose that a new predicate $0 \le r$ has been discovered in Step 3. Then we go back to Step 1, and obtain the following refined version of the abstract program:

$$
\begin{aligned}
&rand\ \ k = \\
&\quad \exists(f(\texttt{false},\texttt{false})\,k, f(\texttt{true},\texttt{false})\,k, f(\texttt{true},\texttt{true})\,k) \\
&f\ (b_1, b_2)\ k = \texttt{if}\ b_1\ \texttt{then}\ B(k(b_1, b_2))\ \texttt{else}\ A(k(b_1, b_2)) \\
&randpos\ \ k = rand\ (\lambda(b_1, b_2). \\
&\qquad\qquad \texttt{if}\ b_2\ \texttt{then}\ k(b_1, b_2)\ \texttt{else}\ randpos\ \ k) \\
&\textbf{main} = randpos\ (\lambda x.\texttt{End}).
\end{aligned}
$$

Now each integer value has been abstracted to a *pair* of Booleans b_1, b_2, which represent whether the original value r satisfies $0 \le r$ and $0 < r$ respectively. The tree generated by the new abstract program is the regular tree Y defined by:

$$Y = \exists(A(Y), B(Y), B(\texttt{End})).$$

By choosing the second branch of each $\exists$-node, we get an infinite event sequence $B^\omega = BBB\cdots$ that is fair under $\{(A, B)\}$. Thus, we can finally conclude that the original program P_3 is *not* fair terminating under $\{(A, B)\}$.

The overall flow outlined above is similar to Kuwahara et al.'s method [22] for disproving plain termination. Each step, however, requires non-trivial extensions of their method.

- First, in Step 1, to describe the condition required for the tree generated by an abstract program, we need alternating parity tree automata (APT; or equi-expressive automata like Streett automata); in the case of Kuwahara et al.'s method [22], (co-)trivial automata [1] were sufficient. As a consequence, in Step 2, we need a higher-order model checker for the full class of APT [10, 25]; in Kuwahara et al.'s method, a trivial automata model checker was sufficient. To our knowledge, this is the first serious application of higher-order model checking for the full class of APT to verification of infinite-data higher-order programs.
- Second, as explained through the example, we need a higher-order model checker capable of generating a higher-order tree grammar as a description of a counterexample. Unfortunately, the existing higher-order model checkers for APT did not have this feature. We have applied a technique called *effective selection* [3, 13, 32] and implemented the counterexample genera-

[2] The tree generated by an abstract program is not necessarily regular in general.

$$P \text{ (programs)} ::= \{f_i\ \widetilde{x}_i = e_i\}_{i\in\{1\dots n\}}$$
$$e \text{ (expressions)} ::= () \mid \textbf{event } A;\ e \mid y\ \widetilde{v} \mid \texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x = a \texttt{ in } e \mid \texttt{let } x = *_{\text{int}} \texttt{ in } e$$
$$a \text{ (simple expressions)} ::= n \mid x \mid \mathbf{op}(\widetilde{a})$$
$$v \text{ (values)} ::= n \mid y\ \widetilde{v}$$

$$\frac{f\ \widetilde{x} = e \in P \qquad |\widetilde{x}| = |\widetilde{v}|}{f\ \widetilde{v} \xrightarrow{\epsilon}_P [\widetilde{v}/\widetilde{x}]e} \quad \text{(E-APP)}$$

$$\frac{[\![a]\!] = n}{\texttt{let } x = a \texttt{ in } e \xrightarrow{\epsilon}_P [n/x]e} \quad \text{(E-LET)}$$

$$\frac{n \neq 0}{\texttt{if } n \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\epsilon}_P e_1} \quad \text{(E-IFT)}$$

$$\texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\epsilon}_P e_2 \quad \text{(E-IFF)}$$

$$\texttt{let } x = *_{\text{int}} \texttt{ in } e \xrightarrow{\epsilon}_P [n/x]e \quad \text{(E-RAND)}$$

$$\textbf{event } A;\ e \xrightarrow{A}_P e \quad \text{(E-EV)}$$

Figure 4: The syntax and operational semantics of the source language.

tion feature. The technique of effective selection has only been investigated in a theoretical community (without much consideration on the practicality); ours is the first realistic implementation of effective selection.

- Third, in Step 3, we need to discover predicates from (grammar descriptions of) *infinite* counterexample trees; In Kuwahara et al.'s method, counterexamples were finite trees.

We have formalized the overall method, implemented an automated tool for disproving fair termination, and confirmed the effectiveness of the approach through experiments.

The rest of this paper is structured as follows. Section 2 introduces the language used as the target of our verification method. Sections 3–5 describe each step of our method. Section 6 reports implementation and experiments. Section 7 discusses related work and Section 8 concludes the paper.

2. Language

This section defines the target language of our verification method, and defines the notion of fair termination formally.

The language is a simply typed, call-by-value higher-order functional language. The syntax of the language is shown in Figure 4. In the figure, $\widetilde{x}$ abbreviates a sequence $x_1, \dots, x_k$ (similarly for $\widetilde{v}$ and $\widetilde{a}$); we write $|\widetilde{x}|$ for the length of the sequence $\widetilde{x}$. A program consists of a finite set of mutually recursive function definitions. We assume that there exists i such that $f_i = \textbf{main}$, which is the "main" function, and $|\widetilde{x}_i| = 0$. We call $|\widetilde{x}_i|$ the arity of function f_i. The expression () represents the unit value. For the sake of simplicity, we assume that programs are represented in the CPS style; thus every function returns the unit value. The expression **event** $A;\ e$ raises an event A, and then evaluates e. We assume a finite set of events, and use meta-variables A and B for them. Event expressions are just used for signaling that certain control points have been reached; as demonstrated in Section 1, they are useful for specifying program properties. The expression $y\ \widetilde{v}$ represents a function call, where y may be a variable or a function name f_i. The value v_i that may be passed to y is either an integer, a variable (the special case of $y\ \widetilde{v}$ where $|\widetilde{v}| = 0$), or a partial application (thus, $|\widetilde{v}|$ must be smaller than the arity of y). The conditional expression $\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2$ evaluates e_2 if the value of a is 0, and e_1 otherwise. We sometimes write `true` for 1 and `false` for 0. The meta-variable a ranges over the set of *simple* expressions, which consists of integers (ranged over by the meta-variable n), variables, and integer operators. The meta-variable $\mathbf{op}$ ranges over a set of integer operators; we assume that we have standard primitives like $+, =, <$, and $\leq$ (where $=, <, \leq$ are assumed to return 0 or 1, based on the interpretation of Booleans above). The expression $\texttt{let } x = *_{\text{int}} \texttt{ in } e$ picks an integer in a non-deterministic manner, binds x to it, and evaluates e.[3]

The operational semantics of the language is given via a labeled reduction relation $e \xrightarrow{l}_P e'$, which is defined in Figure 4. In the figure, $[\![a]\!]$ stands for the value of the simple expression a. Label l is either an event symbol or the empty sequence ϵ. We write $e \xrightarrow{s}^*_P e'$ if $e \xrightarrow{l_1}_P \cdots \xrightarrow{l_n}_P e'$ with $s = l_1 \cdots l_n$. We omit the subscript P if it is clear from the context.

We consider only well-typed programs. The syntax of types and the typing rules are given in Figure 5. The type $\mathbf{int}$ and $\star$ describe integers and the unit value, respectively. The type $\tau_1 \to \tau_2$ describes functions from τ_1 to τ_2. We assume $\tau_1 \neq \star$ in $\tau_1 \to \tau_2$. In typing rules, we abbreviate $\tau_1 \to \cdots \to \tau_n \to \star$ and $\widetilde{x_1} : \widetilde{\tau_1}, \dots, \widetilde{x_n} : \widetilde{\tau_n}$ to $\widetilde{\tau} \to \star$ and $\widetilde{x_i} : \widetilde{\tau_i}$, respectively. The typing rules are standard, except that the body of each function definition must have type $\star$. This requirement is imposed for the sake of convenience for formalizing the predicate abstraction in the next section. Note that we do not lose generality because we can apply continuation-passing-style (CPS) transformation to ensure that every function satisfies the requirement; the type $\star$ corresponds to the answer type in CPS. These restrictions are just for the technical convenience for formalizing the method; the actual implementation reported in Section 6 does not impose such restrictions.

Example 1. The program P_3 in Section 1 is expressed as the following program P'_3 in our language:

$$\begin{array}{l}
\mathit{rand}\ k = \texttt{let } r = *_{\text{int}} \texttt{ in } f\ r\ k \\
f\ r\ k = \texttt{if } 0 \leq r \texttt{ then } (\textbf{event } B;\ k\ r) \texttt{ else } (\textbf{event } A;\ k\ r) \\
\mathit{randpos}\ k = \mathit{rand}\ (g\ k) \\
g\ k\ x = \texttt{if } 0 < x \texttt{ then } k\, x \texttt{ else } \mathit{randpos}\ k \\
\textbf{main} = \mathit{randpos}\ h \\
h\ x = ().
\end{array}$$

For readability, we sometimes use λ-abstractions (as in Section 1). The main function **main** is reduced as follows.

$$\textbf{main} \xrightarrow{\epsilon}_{P'_3} \mathit{randpos}\, h \xrightarrow{\epsilon}_{P'_3} \mathit{rand}\ (g\, h) \xrightarrow{\epsilon}^*_{P'_3} f\ 0\ (g\, h) \xrightarrow{B}^*_{P'_3} g\, h\, 0 \xrightarrow{\epsilon}^*_{P'_3} \mathit{randpos}\, h \xrightarrow{B}^*_{P'_3} \mathit{randpos}\, h \xrightarrow{B}^*_{P'_3} \cdots$$

Thus, the program has an infinite execution sequence labeled by $B^\omega = BBB\cdots$. □

We now define the notion of fair termination.

[3] Thus, contrary to the explanation in Section 1, we do not assume the randomness or fairness of $*_{\text{int}}$ a priori. That $*_{\text{int}}$ returns values in a fair manner should be declared using events and fairness constraints as done in Section 1.

Types:

$$\tau ::= \mathbf{int} \mid \star \mid \tau_1 \to \tau_2$$

Typing for expressions:

$$\Delta \vdash_{\mathrm{ST}} () : \star$$

$$\frac{\Delta \vdash_{\mathrm{ST}} e : \star}{\Delta \vdash_{\mathrm{ST}} \mathbf{event}\ A;\ e : \star}$$

$$\frac{\Delta(y) = \tau_1 \to \cdots \to \tau_k \to \tau \quad \Delta \vdash_{\mathrm{ST}} v_i : \tau_i \quad (\text{for all } i \in \{1, \ldots, k\})}{\Delta \vdash_{\mathrm{ST}} y\ v_1\ \cdots\ v_k : \tau}$$

$$\frac{\Delta \vdash_{\mathrm{ST}} a : \mathbf{int} \qquad \Delta \vdash_{\mathrm{ST}} e_1 : \star \qquad \Delta \vdash_{\mathrm{ST}} e_2 : \star}{\Delta \vdash_{\mathrm{ST}} \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \star}$$

$$\frac{\Delta \vdash_{\mathrm{ST}} a : \mathbf{int} \qquad \Delta, x : \mathbf{int} \vdash_{\mathrm{ST}} e : \star}{\Delta \vdash_{\mathrm{ST}} \texttt{let}\ x = a\ \texttt{in}\ e : \star}$$

$$\frac{\Delta, x : \mathbf{int} \vdash_{\mathrm{ST}} e : \star}{\Delta \vdash_{\mathrm{ST}} \texttt{let}\ x = *_{\mathrm{int}}\ \texttt{in}\ e : \star}$$

Typing for simple expressions:

$$\Delta \vdash_{\mathrm{ST}} n : \mathbf{int}$$

$$\frac{\Delta \vdash_{\mathrm{ST}} a_i : \mathbf{int} \quad (\text{for each } i \in \{1, \ldots, \mathtt{ar}(\mathbf{op})\})}{\Delta \vdash_{\mathrm{ST}} \mathbf{op}(a_1, \ldots, a_k) : \mathbf{int}}$$

Typing for programs:

$$\frac{\Delta = f_1 : \widetilde{\tau_1} \to \star, \ldots, f_k : \widetilde{\tau_k} \to \star \qquad \Delta(\mathbf{main}) = \star \quad \Delta, \widetilde{x_i} : \widetilde{\tau_i} \vdash_{\mathrm{ST}} e_i : \star \quad (\text{for each } i \in \{1, \ldots, k\})}{\vdash_{\mathrm{ST}} \{f_1\ \widetilde{x_1} = e_1, \ldots, f_k\ \widetilde{x_k} = e_k\} : \Delta}$$

Figure 5: Simple type system of the source language.

Definition 2.1 (Fairness constraints)**.** A *(Streett) fairness constraint* $\mathcal{C}$ is a set of event pairs $\{(A_1, B_1), \ldots, (A_n, B_n)\}$.

Definition 2.2 (Fairness)**.** Let $\mathcal{C}$ be a fairness constraint and s be a possibly infinite sequence of events. The sequence s is *fair* with respect to $\mathcal{C}$ when, for every $(A_i, B_i) \in \mathcal{C}$, if A_i occurs infinitely often in s, then so does B_i.

Definition 2.3 (Fair Termination)**.** Let $\mathcal{C}$ be a fairness constraint and P be a program. P is *fair terminating* under $\mathcal{C}$ if there exists no infinite reduction sequence

$$\mathbf{main} \xrightarrow{l_1}_P e_1 \xrightarrow{l_2}_P e_2 \xrightarrow{l_3}_P \cdots,$$

such that the event sequence $l_1 l_2 l_3 \cdots$ is fair with respect to $\mathcal{C}$.

In other words, a program is fair terminating if every non-terminating reduction sequence is unfair. Note that plain termination is a special case of fair termination, where the fairness constraint $\mathcal{C}$ is empty.

Example 2. Recall the program P_3' in Example 1. It is *not* fair terminating under $\{(A, B)\}$ because it has a fair infinite reduction sequence:

$$\mathbf{main} \xrightarrow{B}{}^{*}_{P_3'} \mathit{randpos}\ h \xrightarrow{B}{}^{*}_{P_3'} \mathit{randpos}\ h \xrightarrow{B}{}^{*}_{P_3'} \cdots$$

Let P_4 be the program obtained from P_3' by replacing the condition $0 < x$ in the function g with $0 \leq x$. P_4 is fair terminating, as every infinite reduction sequence must be of the form:

$$\mathbf{main} \xrightarrow{A}{}^{*}_{P_4} \mathit{randpos}\ h \xrightarrow{A}{}^{*}_{P_4} \mathit{randpos}\ h \xrightarrow{A}{}^{*}_{P_4} \cdots,$$

which is not fair. □

The goal of the present paper is to develop an automated method for disproving the fair termination of a given program under a given fairness constraint. In other words, we develop a method for verifying that a given program has a fair infinite reduction sequence.

3. Step 1: Reduction to Higher-Order Model Checking

The task of Step 1 is to convert a given program P and a fairness constraint $\mathcal{C}$ to an abstract program D_P and a tree automaton $\mathcal{A}_\mathcal{C}$ respectively. The former generates a tree that abstracts possible execution traces of the program, and the latter describes a sufficient condition on the tree generated by D_P in order for P to have a fair non-terminating execution trace. The construction of D_P is actually almost the same as Kuwahara et al.'s work [22], except that we have event primitives. To make the paper self-contained, however, we briefly describe it in Section 3.1. The construction of $\mathcal{A}_\mathcal{C}$ is new, which is described in Section 3.2.

3.1 Construction of the Abstract Program D_P

In addition to a source program, we assume that we are given predicates used for abstracting each integer. They are supplied by Step 3; initially, the set of predicates is empty. The predicates used for abstraction are actually specified in the form of types called *abstraction types*, so that we can use different predicates for each position in the source program. The set of abstraction types is given by:

σ (abstraction types) $::= \star \mid \mathbf{int}[Q_1, \ldots, Q_k] \mid x : \sigma_1 \to \sigma_2$
Q (predicates) $::= \lambda x.\varphi$
φ (formulas) $::= n_1 x_1 + \cdots + n_k x_k \leq n \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi.$

The type $\star$ is the unit type and, the type $\mathbf{int}[Q_1, \ldots, Q_k]$ describes integers n that should be abstracted to a tuple of Booleans $(Q_1(n), \ldots, Q_k(n))$. For example, the integer 3 is abstracted by the abstraction type $\mathbf{int}[\lambda x.\ x = 0, \lambda x.\ x > 0]$ to $(\mathtt{false}, \mathtt{true})$. The dependent function type $x : \sigma_1 \to \sigma_2$ describes functions that takes a value x of type σ_1 and returns a value of type σ_2. When x does not occur in σ_2, we just write $\sigma_1 \to \sigma_2$. We assume that, along with a program P, we are given an abstraction type σ_i for each function f_i in P. For example, for the program P_3' in Example 1, we may be given the following types for functions:

$\mathit{rand} : (\mathbf{int}[\lambda x.0 < x] \to \star) \to \star$
$f : \mathbf{int}[\lambda x.0 < x] \to (\mathbf{int}[\lambda x.0 < x] \to \star) \to \star$
$\mathit{randpos} : (\mathbf{int}[\lambda x.0 < x] \to \star) \to \star$
$g : (\mathbf{int}[\lambda x.0 < x] \to \star) \to \mathbf{int}[\lambda x.0 < x] \to \star$
$\mathbf{main} : \star$
$h : \mathbf{int}[\lambda x.0 < x] \to \star,$

which specifies that every integer should be abstracted using the predicate $\lambda x.0 < x$.

We use the language given in Figure 6 for describing abstract programs. It is a simply-typed, higher-order functional language having Booleans, tuples, and tree constructors as primitive data constructors. As in the source language, a program consists of a

finite set of mutually recursive function definitions; we assume it contains a definition of the main function **main**. The language has five kinds of tree constructors: A (for each event name A), `Call`, `End`, $\forall_i$, and $\exists_i$. The constructor A and `Call` are unary; they are used to express the occurrences of event A and a function call respectively. The constructor `End` is nullary, and used to express the termination of the source program P. The i-ary constructor $\forall_i$ is used to express that only one of the branches is guaranteed to correspond to an execution of the source program P; the other branches may be spurious. Thus, for the purpose of proving the existence of a fair infinite execution, it suffices to show that *all* the branches have fair infinite paths, hence the name of the tree constructor "$\forall$". On the other hand, the i-ary constructor $\exists_i$ is used to express that every branch corresponds to an execution of the source program P. Thus, it suffices to show that *one of* the branches has a fair infinite path.

The expression $\forall\{\psi_1 \to M_1, \dots, \psi_k \to M_k\}$ ($\exists\{\psi_1 \to M_1, \dots, \psi_k \to M_k\}$, resp.) generates a node with label $\forall$ ($\exists$, resp.), and for each $i \in \{1, \dots, k\}$, evaluates ψ_i and add the tree generated by M_i as a child of the node only if the value of ψ_i is `true`. The Boolean expression $\sharp_i(x)$ is the i-th projection of the tuple x. For example, if $x_1 = (\mathtt{true})$ and $x_2 = (\mathtt{true}, \mathtt{false})$, then $\forall\{\sharp_1(x_1) \to A(\mathtt{End}), \sharp_2(x_2) \to B(\mathtt{End}), \sharp_1(x_1) \vee \sharp_2(x_2) \to \mathtt{End}\}$ evaluates to $\forall_2(A(\mathtt{End}), \mathtt{End})$. When x is a singleton tuple, we often omit $\sharp_1$ and just write x for $\sharp_1(x)$. The meaning of the other expressions should be clear.

The construction of D_P is formalized as the type-based transformation relations $\Gamma \vdash e : \sigma \rightsquigarrow M$ and $\vdash P : \Gamma \rightsquigarrow D$ defined in Figure 7. The former means that the expression e in the source language can be abstracted to the expression M under the assumption that each free variable x is abstracted according to the abstraction type $\Gamma(x)$, and the latter means that the source program P should be abstracted to D under the assumption that each function f should be abstracted according to the abstraction type $\Gamma(f)$. In the figure, $\models \varphi$ means that φ is a tautology. The expression $b_i Q_i(x)$ stands for $Q_i(x)$ if b_i is `true` and $\neg Q_i(x)$ otherwise. The expression θ_Γ represents the substitution that replaces each variable x of type $\mathbf{int}[Q_1, \dots, Q_n]$ in Γ with $(Q_1(x), \dots, Q_n(x))$. For example, let Γ be $x : \mathbf{int}[\lambda x.\ x \geq 0], y : \mathbf{int}[\lambda y.\ y = 0, \lambda y.\ y > x]$, then $\theta_\Gamma(\sharp_1(x) \vee \sharp_2(y)) = \sharp_1(x \geq 0) \vee \sharp_2(y = 0, y > x) = x \geq 0 \vee y > x$.

The rule PA-EVENT just turns an event constructor A into the tree constructor. Since the other transformation rules are the same as Kuwahara et al.'s ones [22], we briefly explain only the key rules: PA-IF and PA-RAND. The rule PA-IF *over*-approximates the computation of a source program, since, due to an abstraction, we may not know which branch is actually taken. For example, consider the conditional expression $\mathtt{if}\ 0 < x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$, and suppose that the abstraction type of x is $\mathbf{int}[\lambda x.0 \leq x]$. Then, if the abstraction of x is `false`, we know that $0 < x$ is `false`, so that only the else branch is taken. However, if the abstraction of x is `true`, we only know that the value of x is zero or positive, so that both branches are possible. We thus convert the conditional expression above to:

$$\forall\{x \to M_1, \mathtt{true} \to M_2\},$$

where M_i is an abstraction of e_i. The expression evaluates to $\forall_1(M_2)$ if x is `false`, and $\forall_2(M_1, M_2)$ otherwise. In this way, a deterministic computation step of a source program may be over-approximated by non-deterministic branches, but it is maintained that one of the branches corresponds to the actual computation.

In contrast, the rule PA-RAND *under*-approximates the computation of a source program. That is because, due to an abstraction, we may not know whether some branch exists. For example, consider the expression $\mathtt{let}\ x : \mathbf{int}[\lambda x.0 \leq x < y] = *_{\mathrm{int}}\ \mathtt{in}\ e$.

$$\begin{aligned}
D \text{ (programs)} &::= \{f_i\ \widetilde{x}_i = M_i\}_{i \in \{1 \dots n\}} \\
M \text{ (expressions)} &::= c(M_1, \dots, M_k) \mid y\ \widetilde{V} \\
&\quad \mid \mathtt{let}\ x = (b_1, \dots, b_k)\ \mathtt{in}\ M \\
&\quad \mid \forall\{\psi_1 \to M_1, \dots, \psi_k \to M_k\} \\
&\quad \mid \exists\{\psi_1 \to M_1, \dots, \psi_k \to M_k\} \\
b \text{ (Booleans)} &::= \mathtt{true} \mid \mathtt{false} \\
V \text{ (values)} &::= (b_1, \dots, b_k) \mid y\ \widetilde{V} \\
c \text{ (tree constructors)} &::= A \mid \mathtt{Call} \mid \mathtt{End} \mid \forall_i \mid \exists_i \\
\psi \text{ (Boolean exp.)} &::= b \mid \sharp_i(x) \mid \psi_1 \vee \psi_2 \mid \neg\psi \\
E \text{ (eval. context)} &::= [\,] \\
&\quad \mid c(M_1, \dots, M_{i-1}, E, M_{i+1}, \dots, M_n)
\end{aligned}$$

$$E[\mathtt{let}\ x = (b_1, \dots, b_k)\ \mathtt{in}\ M] \to_D E[[(b_1, \dots, b_k)/x]M]$$

$$\frac{f\ x_1 \dots x_k = M \in D}{E[f\ V_1 \cdots V_k] \to_D E[[V_1/x_1, \dots, V_k/x_k]M]}$$

$$\frac{\{\psi_i \mid i \in \{1, \dots, k\}, [\![\psi_i]\!] = \mathtt{true}\} = \{\psi_{i_1}, \dots, \psi_{i_\ell}\}}{E[\forall\{\psi_1 \to M_1, \dots, \psi_k \to M_k\}] \to_D E[\forall_\ell(M_{i_1}, \dots, M_{i_\ell})]}$$

$$\frac{\{\psi_i \mid i \in \{1, \dots, k\}, [\![\psi_i]\!] = \mathtt{true}\} = \{\psi_{i_1}, \dots, \psi_{i_\ell}\}}{E[\exists\{\psi_1 \to M_1, \dots, \psi_k \to M_k\}] \to_D E[\exists_\ell(M_{i_1}, \dots, M_{i_\ell})]}$$

Figure 6: The syntax and operational semantics of the language of abstract programs.

The abstraction type $\mathbf{int}[\lambda x.0 \leq x < y]$ specifies that the random number x should be abstracted to the Boolean value representing whether $0 \leq x < y$ holds. Thus, we basically replace the generation of a random integer with that of a random Boolean. However, depending on the actual value of y, we may not know whether $0 \leq x < y$ can be `true`. Thus, for example, if the abstraction type of y is $\mathbf{int}[\lambda y.2 < y]$, then we abstract the expression to:

$$\exists\{y \to \mathtt{let}\ x = \mathtt{true}\ \mathtt{in}\ M, \mathtt{true} \to \mathtt{let}\ x = \mathtt{false}\ \mathtt{in}\ M\},$$

where M is an abstraction of e. Thus, the branch for $x = \mathtt{true}$ is created only when (the abstract value of) y is `true` (which means that $2 < y$ holds in the original computation). The expression evaluates to $\exists_2(\mathtt{let}\ x = \mathtt{true}\ \mathtt{in}\ M, \mathtt{let}\ x = \mathtt{false}\ \mathtt{in}\ M)$ if $y = \mathtt{true}$, and $\exists_1(\mathtt{let}\ x = \mathtt{false}\ \mathtt{in}\ M)$ otherwise.

Example 3. Recall the program P_3' in Example 1. Let Γ be

$$\begin{aligned}
&\mathit{rand} : (\mathbf{int}[\lambda x.\ 0 < x] \to \star) \to \star \\
&f : \mathbf{int}[\lambda x.\ 0 < x] \to (\mathbf{int}[\lambda x.0 < x] \to \star) \to \star \\
&g : (\mathbf{int}[\lambda x.\ 0 < x] \to \star) \to \mathbf{int}[\lambda x.0 < x] \to \star \\
&\mathit{randpos} : (\mathbf{int}[\lambda x.0 < x] \to \star) \to \star \\
&\mathbf{main} : \star,
\end{aligned}$$

$$\frac{}{\Gamma \vdash () : \star \rightsquigarrow \texttt{End}} \quad \text{(PA-UNIT)}$$

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow M}{\Gamma \vdash \textbf{event } A;\ e : \sigma \rightsquigarrow A(M)} \quad \text{(PA-EVENT)}$$

$$\frac{\begin{array}{c}\Gamma, x : \textbf{int}[Q_1, \ldots, Q_k] \vdash e : \star \rightsquigarrow M \\ \models b_1 Q_1(a) \wedge \cdots \wedge b_k Q_k(a) \Rightarrow \theta_\Gamma \psi_{(b_1,\ldots,b_k)} \\ \text{(for each } b_1, \ldots, b_k \in \{\texttt{true}, \texttt{false}\})\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let } x : \textbf{int}[Q_1, \ldots, Q_k] = a \texttt{ in } e : \star \rightsquigarrow \\ \forall \left\{ \begin{array}{l} \psi_{(b_1,\ldots,b_k)} \rightarrow \\ \texttt{let } x = (b_1, \ldots, b_k) \texttt{ in } M \end{array} \middle| \ b_1, \ldots, b_k \in \{\texttt{true}, \texttt{false}\} \right\}\end{array}} \quad \text{(PA-SEXP)}$$

$$\frac{\begin{array}{cc}\models a \neq 0 \Rightarrow \theta_\Gamma \psi_1 & \models a = 0 \Rightarrow \theta_\Gamma \psi_2 \\ \Gamma \vdash e_1 : \star \rightsquigarrow M_1 & \Gamma \vdash e_2 : \star \rightsquigarrow M_2\end{array}}{\Gamma \vdash \texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2 : \star \rightsquigarrow \forall\{\psi_1 \rightarrow M_1, \psi_2 \rightarrow M_2\}} \quad \text{(PA-IF)}$$

$$\frac{\begin{array}{c}\Gamma, x : \textbf{int}[Q_1, \ldots, Q_k] \vdash e : \star \rightsquigarrow M \\ \models \theta_\Gamma \psi_{(b_1,\ldots,b_k)} \Rightarrow \exists x. b_1 Q_1(x) \wedge \cdots \wedge b_k Q_k(x) \\ \text{(for each } b_1, \ldots, b_k \in \{\texttt{true}, \texttt{false}\})\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let } x : \textbf{int}[Q_1, \ldots, Q_k] = *_{\text{int}} \texttt{ in } e : \star \rightsquigarrow \\ \exists \left\{ \begin{array}{l} \psi_{(b_1,\ldots,b_k)} \rightarrow \\ \texttt{let } x = (b_1, \ldots, b_k) \texttt{ in } M \end{array} \middle| \ b_1, \ldots, b_k \in \{\texttt{true}, \texttt{false}\} \right\}\end{array}} \quad \text{(PA-RAND)}$$

$$\frac{\begin{array}{c}\Gamma(y) = x_1 : \sigma_1 \rightarrow \cdots \rightarrow x_k : \sigma_k \rightarrow \sigma \\ \Gamma \vdash v_i : [v_1/x_1, \ldots, v_{i-1}/x_{i-1}]\sigma_i \rightsquigarrow V_i \\ \text{(for each } i \in \{1, \ldots, k\})\end{array}}{\Gamma \vdash y\ v_1 \cdots v_k : [v_1/x_1, \ldots, v_k/x_k]\sigma \rightsquigarrow y\ V_1 \cdots V_k} \quad \text{(PA-APP)}$$

$$\frac{\begin{array}{c}\{f_i\ \widetilde{x} : \widetilde{\sigma_i} \rightarrow \star\}_{i \in \{1,\ldots,k\}}, \widetilde{x} : \widetilde{\sigma_j} \vdash e_i : \star \rightsquigarrow M_i \\ \text{(for each } j \in \{1, \ldots, k\})\end{array}}{\begin{array}{c}\vdash \{f_i \widetilde{x_i} = e_i\}_{i \in \{1,\ldots,k\}} : \{f_i : \widetilde{x} : \widetilde{\sigma_i} \rightarrow \star\}_{i \in \{1,\ldots,k\}} \rightsquigarrow \\ \{f_i\ \widetilde{x_i} = \texttt{Call}(M_i)\}_{i \in \{1,\ldots,k\}}\end{array}} \quad \text{(PA-PROG)}$$

Figure 7: Predicate abstraction rules.

then P'_3 is abstracted to the following program $D_{P'_3}$ (i.e., $\vdash P'_3 : \Gamma \rightsquigarrow D_{P'_3}$):

$$\begin{array}{rl} rand\ k = & \texttt{Call}(\exists\{\texttt{true} \rightarrow \texttt{let } r = \texttt{false in } f\ r\ k, \\ & \qquad \texttt{true} \rightarrow \texttt{let } r = \texttt{true in } f\ r\ k\}) \\ f\ b_{0<r}\ k = & \texttt{Call}(\forall\{\texttt{true} \rightarrow B(k(b_{0<r})), \\ & \qquad \neg b_{0<r} \rightarrow A(k(b_{0<r}))\}) \\ randpos\ k = & \texttt{Call}(rand\ (g\ k)) \\ g\ k\ b_{0<x} = & \texttt{Call}(\forall\{b_{0<x} \rightarrow k\ b_{0<x}, \neg b_{0<x} \rightarrow randpos\ k\}) \\ \textbf{main} = & \texttt{Call}(randpos\ (\lambda b_{0<x}.\ \texttt{End})). \end{array}$$

Here, we use type $\textbf{int}[\lambda r.\ 0 < r]$ for the type of r in the body of $rand$. The constructor `Call` has been inserted to the body of each function definition, according to rule PA-PROG. This ensures that every infinite reduction sequence has an infinite sequence of labels. The tree generated by $D_{P'_3}$ is shown in Figure 8; we have omitted `Call` nodes for the sake of simplicity. □

3.2 Construction of the Tree Automaton $\mathcal{A}_\mathcal{C}$

We now construct a tree automaton $\mathcal{A}_\mathcal{C}$, which expresses a sufficient condition on the tree generated by D_P for P to have a fair infinite execution trace. As already explained, the condition is, informally, that (i) every branch of $\forall$-node has an fair infinite path, and (ii) at least one of the branches of $\exists$-node has an fair infinite

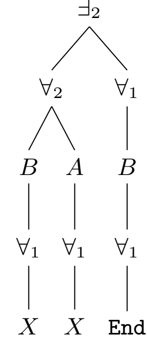

Figure 8: The tree generated by $D_{P'_3}$. Here X is the same as the whole tree.

path. That condition can be easily expressed by a Streett automaton (for infinite trees). For the sake of simplicity, we assume below that the arities of all the $\forall$ and $\exists$ nodes are 2; note that $\forall_i(T_1, \ldots, T_i)$ can be replaced by $\forall_2(T_1, \forall_2(T_2, \forall_2(\cdots \forall_2(T_{i-1}, T_i) \cdots)))$.

We first review the definition of Streett automata. We write $\mathcal{P}(X)$ for the powerset of X.

Definition 3.1 (Σ-labeled trees)**.** A *tree* is a set f of sequences of positive integers, such that whenever $\pi i \in f$, $\{\pi\} \cup \{\pi j \mid 1 \leq j < i\} \subseteq f$. Let Σ be a ranked alphabet, i.e., a map from a finite set of symbols to the set of non-negative integers (called arities). A Σ*-labeled tree* is a map T from a tree to $dom(\Sigma)$, such that for every $\pi \in dom(T)$, $\{i \mid \pi i \in dom(T)\} = \{1, \ldots, \Sigma(T(\pi))\}$.

Definition 3.2 (Streett automata)**.** Let Σ be a ranked alphabet. A non-deterministic Streett tree automaton is a quintuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, \mathcal{C})$ where

- Σ is a ranked alphabet,
- Q is a finite set of states,
- δ, called a *transition function*, is a map from $Q \times dom(\Sigma)$ to $\mathcal{P}(Q^*)$. Whenever $q_1 \cdots q_\ell \in \delta(q, a)$, it must be the case that $\ell = \Sigma(a)$.
- $q_0 \in Q$ is an initial state, and
- $\mathcal{C}$, called a Streett acceptance condition, is of the form $\{(E_1, F_1), \ldots, (E_k, F_k)\}$, where $E_i, F_i \subseteq Q$.

A *run-tree* R of $\mathcal{A}$ over a Σ-labeled tree T is a tree obtained from T by annotating each node of T with an element of Q, so that

- The root node is annotated with q_0
- If a node with label a is annotated with q, then there must exist $q_1 \cdots q_\ell \in \delta(q, a)$ such that the i-th child of the node is annotated with q_i for each $i \in \{1, \ldots, \ell\}$.

A run-tree R is *accepting* if, for every infinite path π of R and every $i \in \{1, \ldots, k\}$, if an element of E_i occurs infinitely often in π, then some element of F_i must also occur infinitely often in π. We write $Lang(\mathcal{A})$ for a set of trees accepted by $\mathcal{A}$.

We can now define $\mathcal{A}_\mathcal{C}$ as a Streett automaton for a fairness constraint $\mathcal{C}$.

Definition 3.3 ($\mathcal{A}_\mathcal{C}$)**.** Let $\textbf{Ev}$ be the finite set of event names that occur in a given program, and let $\mathcal{C} = \{(A_1, B_1), \cdots, (A_n, B_n)\}$ (where $\{A_1, B_1, \ldots, A_n, B_n\} \subseteq \textbf{Ev}$) be a fairness constraint. A Streett tree automaton $\mathcal{A}_\mathcal{C}$ is defined by:

$$\mathcal{A}_\mathcal{C} = (\Sigma, \{q_0, q_\perp\} \cup \{q_A \mid A \in \textbf{Ev}\}, \delta, q_0, \mathcal{C})$$

where

$$\Sigma = \{\texttt{End} \mapsto 0, \texttt{Call} \mapsto 1, \forall \mapsto 2, \exists \mapsto 2\} \cup \{A \mapsto 1 \mid A \in \mathbf{Ev}\}$$

$$\delta(q, \texttt{End}) = \begin{cases} \{\epsilon\} & \text{if } q = q_\bot \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta(q, \texttt{Call}) = \begin{cases} \{q_\bot\} & \text{if } q = q_\bot \\ \{q_0\} & \text{otherwise} \end{cases}$$

$$\delta(q, \forall) = \begin{cases} \{q_\bot q_\bot\} & \text{if } q = q_\bot \\ \{q_0 q_0\} & \text{otherwise} \end{cases}$$

$$\delta(q, \exists) = \begin{cases} \{q_\bot q_\bot\} & \text{if } q = q_\bot \\ \{q_0 q_\bot, q_\bot q_0\} & \text{otherwise} \end{cases}$$

$$\delta(q, A) = \begin{cases} \{q_\bot\} & \text{if } q = q_\bot \\ \{q_A\} & \text{otherwise} \end{cases}$$

$$\mathcal{C} = \{(\{q_{A_1}\}, \{q_{B_1}\}), \ldots, (\{q_{A_n}\}, \{q_{B_n}\})\}.$$

The automaton just sets the next state to the event label of the current node, and then checks that the fairness condition holds by using the acceptance condition. Upon visiting a $\forall$-node, the automaton checks every branch, whereas upon visiting a $\exists$-node, it picks only one of branches, and ignores the other branch (with state $q_\bot$, from which any tree is accepted).

The following theorem states the soundness of our reduction.

Theorem 1 (Soundness)**.** *Let P be a program and $\mathcal{C}$ be a fairness constraint. If* **Tree**$(D_P) \in Lang(\mathcal{A}_\mathcal{C})$*, then P is* NOT *fair terminating under $\mathcal{C}$.*

Proof sketch. Let e be an expression of the source language and suppose that $\vdash P : \Gamma \rightsquigarrow D_P$ and $\Gamma \vdash e : \star \rightsquigarrow M$. Then, the reductions of M can be simulated by those of e in the following sense.

- If $M \longrightarrow^* \forall_k(M_1, \ldots, M_k)$,[4] then there exists i such that $e \xrightarrow{\epsilon}{}^*_P e'$, $\Gamma \vdash e' : \star \rightsquigarrow M'_i$, and $M_i \longrightarrow^* M'_i$ for some e' and M'_i
- If $M \longrightarrow^* \exists_k(M_1, \ldots, M_k)$, then for every $i \in \{1, \ldots, k\}$, there exist e' and M'_i such that $e \xrightarrow{\epsilon}{}^*_P e'$, $\Gamma \vdash e' : \star \rightsquigarrow M'_i$, and $M_i \longrightarrow^* M'_i$.
- If $M \longrightarrow^* A(M')$, then there exist e' and M'' such that $e \xrightarrow{A}{}^*_P e'$, $\Gamma \vdash e' : \star \rightsquigarrow M''$, and $M' \longrightarrow^* M''$.
- If $M \longrightarrow^* \texttt{Call}(M')$, then there exist e' and M'' such that $e \xrightarrow{\epsilon}{}^+_P e'$, $\Gamma \vdash e' : \star \rightsquigarrow M''$, and $M' \longrightarrow^* M''$.

We omit the proof of these facts since it is almost the same as that of the corresponding theorem in [22]. Now, suppose that **Tree**(D_P) is accepted by $\mathcal{A}_\mathcal{C}$, i.e., there exists an accepting run-tree R of $\mathcal{A}_\mathcal{C}$ over **Tree**(D_P). Let $e = M =$ **main**. We can construct a fair infinite execution path of P from the triple (e, M, R) as follows.

- If $R = \forall^q_k(R_1, \ldots, R_k)$, then $M \longrightarrow^* \forall_k(M_1, \ldots, M_k)$, where R_i is a run-tree for the tree generated by M_i, and the root of each R_i is annotated with q_0. By the simulation property above, there must be a corresponding execution sequence $e \xrightarrow{\epsilon}{}^*_P e'$ such that e' "simulates" M_i. Thus, we continue the construction for (e', M_i, R_i).
- If $R = \exists^q_k(R_1, \ldots, R_k)$, then $M \longrightarrow^* \exists_k(M_1, \ldots, M_k)$ where R_i is a run-tree for the tree generated by M_i. By the construction of $\mathcal{A}_\mathcal{C}$, there exists i such that the root of R_i is annotated with q_0. By the simulation property above, there must be a corresponding execution sequence $e \xrightarrow{\epsilon}{}^*_P e'$ such that e' simulates M_i. Thus, we continue the construction for (e', M_i, R_i).
- If $R = A^q(R_1)$, then $M \longrightarrow^* A(M_1)$ where R_1 is a run-tree for the tree generated by M_1, and the root of R_1 is annotated with q_A. By the simulation property above, there must exist e' such that $e \xrightarrow{A}{}^*_P e'$ and e' simulates M_1.
- If $R = \texttt{Call}^q(R_1)$, then $M \longrightarrow^* \texttt{Call}(M_1)$ where R_1 is a run-tree for the tree generated by M_1, and the root of R_1 is annotated with q_0. By the simulation property above, there must exist e' such that $e \longrightarrow^+_P e'$ and e' simulates M_1.

Since R is an accepting run-tree, we can continue the process above indefinitely and obtain an infinite execution sequence

$$\textbf{main} \xrightarrow{A_1}{}^*_P e_1 \xrightarrow{A_2}{}^*_P e_2 \xrightarrow{A_3}{}^*_P e_3 \xrightarrow{A_4}{}^*_P \cdots.$$

Since R satisfies the acceptance condition, the sequence of events $A_1 A_2 A_3 A_4 \cdots$ must be fair. □

4. Step 2: Higher-Order Model Checking

The task of Step 2 is, given an abstract program D_P and a tree automaton $\mathcal{A}_\mathcal{C}$ produced in Step 1, to check whether the tree generated by D_P (i.e., **Tree**(D_P)) is accepted by $\mathcal{A}_\mathcal{C}$, and generate a counterexample if the answer is negative. Whether **Tree**$(D_P) \in Lang(\mathcal{A}_\mathcal{C})$ holds can be decided by using an existing APT higher-order model checker [9, 10, 25], but we need to add the functionality to generate a counterexample. Below we briefly review higher-order model checking [28] and explain how to use it to decide **Tree**$(D_P) \stackrel{?}{\in} Lang(\mathcal{A}_\mathcal{C})$ in Section 4.1. We then describe the new method for counterexample generation in Section 4.2.

4.1 Higher-Order Model Checking

Higher-order model checking is the problem of deciding whether the tree generated by a given higher-order recursion scheme (HORS) is accepted by a given tree automaton. A HORS is essentially a simply-typed, tree-generating higher-order functional program. It consists of the following mutually recursive function definitions:

$$\begin{array}{l} \textbf{main} = t_0 \\ f_1\, x_{1,1} \cdots x_{1,k_1} = t_1 \\ \cdots \\ f_n\, x_{n,1} \cdots x_{n,k_n} = t_n \end{array}$$

where **main** is the main function (or the start symbol, in the terminology of formal grammars), and t_i is an applicative term constructed from variables, functions, and tree constructors. Its syntax is given by:

$$t ::= x \mid f \mid A \mid t_1 t_2,$$

where A ranges over a finite set of tree constructors.[5] It is decidable whether the tree generated by a HORS is accepted by a tree automaton [28], and a few model checkers are available that support alternating parity tree automata (APT) [9, 10, 25]. Thus, we just need to convert D_P and $\mathcal{A}_\mathcal{C}$ to a HORS and an APT.

An abstract program can be easily converted to HORS, by applying Church encoding to Boolean values. For example, the

[4] Here, for the sake of simplicity, we assume $\mathcal{A}_\mathcal{C}$ is extended to handle k-ary $\forall$ and $\exists$ symbols directly, without the encoding using binary symbols.

[5] The usual convention of HORS is to use upper letters for function symbols (called non-terminals) and lower letters for tree constructors. We use the opposite (lower letters for function symbols and variables, and upper letters for tree constructors), following the usual convention for functional programs.

program:

$$rand\ k = \texttt{Call}(\exists\{\texttt{true} \to m\,\texttt{true}\,k, \texttt{true} \to m\,\texttt{false}\,k\})$$
$$m\,r\,k = \forall\{\texttt{true} \to B\ (k\ r), \neg r \to A\ (k\ r)\}$$
$$g\ x = \texttt{Call}(\forall\{x \to \texttt{End}, \neg x \to rand\ g\})$$
$$\textbf{main} = rand\ g$$

can be converted to:

$$\textbf{main} = rand\ g$$
$$rand\ k = \texttt{Call}(\exists_2\,(m\,\texttt{true}\,k)\,(m\,\texttt{false}\,k))$$
$$m\,r\,k = \texttt{if}\,r\,(\forall_1\,(B(k\,r)))(\forall_2\,(B(k\,r))\,(A(k\,r)))$$
$$g\,x = \texttt{Call}(\texttt{if}\,x\,(\forall_1\,\texttt{End})\,(\forall_1\,(rand\,g)))$$
$$\texttt{if}\,b\,x\,y = b\,x\,y$$
$$\texttt{true}\,x\,y = x$$
$$\texttt{false}\,x\,y = y.$$

As for tree automata, Streett tree automata and APT are actually equi-expressive, and we can use the standard technique [12] to convert $\mathcal{A}_\mathcal{C}$ to an equivalent APT.

4.2 Counterexample Generation

As already explained, when **Tree**(D_P) is not accepted by $\mathcal{A}_\mathcal{C}$, we need to generate a *counterexample* and pass it to Step 3. By the discussion above, we may assume that D_P and $\mathcal{A}_\mathcal{C}$ are a HORS and an APT respectively.

A counterexample is a part of the tree **Tree**(D_P) that violates the property described by $\mathcal{A}_\mathcal{C}$. More formally, let $\mathcal{A}^\perp$ be the automaton obtained from $\mathcal{A}$ by adding a new nullary symbol $\perp$, which is blindly accepted from any state. A counterexample against **Tree**$(D) \in Lang(\mathcal{A})$ is a minimal[6] tree T obtained by replacing some subtrees of **Tree**(D) with $\perp$, such that $T \notin Lang(\mathcal{A}^\perp)$. In the case of our problem where $D = D_P$ and $\mathcal{A} = \mathcal{A}_\mathcal{C}$, a counterexample is a tree T obtained from **Tree**(D_P) by replacing all but one branch of each $\forall$-node with $\perp$, such that every path of T is either finite or unfair. For example, for the tree (shown in Figure 8) generated by $D_{P'_3}$ in Example 3, a counterexample is the tree obtained by replacing the first branch of every $\forall_2$-node with $\perp$.

Since a counterexample defined above may be infinite in general, we use a HORS (or, a tree-generating functional program) as a finite representation of a counterexample. For example, the counterexample mentioned above is expressed by the following HORS consisting of a single function **main**:

$$\textbf{main} = \exists_2\ (\forall_2\ \perp\ (A\ (\forall_1\ \textbf{main})))\ (\forall_1(B\ (\forall_1\ \texttt{End}))).$$

We have extended `HorSatP` [9] with the functionality to generate a counterexample (in the form of a HORS); the same extension would also be applicable to other higher-order model checkers for APT [10, 25]. The key ideas in our counterexample generation are as follows.

- If **Tree**$(D) \notin Lang(\mathcal{A})$, then **Tree**$(D)$ is accepted by the *complement* $\overline{\mathcal{A}}$ of $\mathcal{A}$, and a counterexample tree is the part of **Tree**(D) visited by $\overline{\mathcal{A}}$ when **Tree**(D) is accepted by $\overline{\mathcal{A}}$.[7]
- A HORS representation of a run-tree of $\overline{\mathcal{A}}$ (which describes how $\overline{\mathcal{A}}$ traverses nodes of **Tree**(D)) can be constructed by a technique called *effective selection* [3, 13, 32]. By modifying it (replacing the label of each node of the run-tree with the symbol read by the corresponding transition of the automaton), we can obtain (a HORS representation of) a counterexample.

[6] With respect to the least compatible relation such that $\perp \leq T$ for every tree T.

[7] To ensure the minimality, we need to assume that $\overline{\mathcal{A}}$ visits as few nodes as possible. Fortunately, for $\mathcal{A}_\mathcal{C}$, it is easy to guarantee this: $\overline{\mathcal{A}_\mathcal{C}}$ should visit only one of the subtrees for each $\forall$-node.

- `HorSatP` (as well as other APT higher-order model checkers [10, 25]) is based on Kobayashi and Ong's type system [18] and constructs a type derivation (more precisely, a winning strategy for the typability game [18]). Actually, the soundness proof of Kobayashi and Ong's type system [18] provides an effective algorithm for converting the type derivation to a run-tree of $\mathcal{A}$, which serves as a realistic algorithm for effective selection.

The detail of the counterexample generation method is deferred to a full version of the paper, as understanding it requires knowledge of APT and Kobayashi and Ong's type system [18].

5. Step 3: Predicate Discovery

The task of Step 3 is, given a counterexample tree produced by Step 2, to find new predicates (or abstraction types, more precisely) to refine the predicate abstraction. Assume that a source program P has a fair non-terminating execution sequence, but that Step 2 produces a counterexample. There are the following three possible cases:

(I) A spurious error path (i.e., an unfair infinite path or a finite path ending with `End` that does not correspond to any actual execution trace of P) has been introduced by an over-approximation. For example, consider the following expression of a source program:

$$\texttt{if}\ b\ \texttt{then}\ fair_nonterminating()\ \texttt{else}\ unfair_nonterminating().$$

Suppose that b always evaluates to `true` in the source program, but we cannot infer so during predicate abstraction. Then, the tree generated by the abstract program is: $\forall_2(T_1, T_2)$, where T_1 has a fair infinite path, but T_2 has only an unfair infinite path. Since we require that both of the branches of $\forall_2$ has a fair infinite path, the tree is rejected and $\forall_2(\perp, T_2)$ is generated as a counterexample. In this case, we need to find a predicate necessary to infer that the condition b is always `true`.

(II) A fair non-terminating execution sequence has been merged with an unfair or terminating one by an approximation of random integers. For example, consider the expression:

$$\texttt{let}\ x : \textbf{int}[\,] = *_{\text{int}}\ \texttt{in}\ \texttt{if}\ 0 < x\ \texttt{then}\ fair_nonterminating()\ \texttt{else}\ unfair_nonterminating().$$

Here, the random number x is abstracted according to an empty set of predicates. Then, the tree generated by the abstract program is: $\exists_1(\forall_2(T_1, T_2))$, where T_1 has a fair infinite path, but T_2 has only an unfair infinite path. Note that no information is available for deciding whether the condition $0 < x$ holds, so the conditional branch has been replaced by the $\forall$-node with both branches. Thus, the tree is rejected, and $\exists_1(\forall_2(\perp, T_2))$ is generated as a counterexample. In this case, we need to add the predicate $\lambda x.0 < x$ to the abstraction type of x. Then, the computations for the case $0 < x$ and the case $x \leq 0$ are split, and the abstract program would generate $\exists_2(\forall_1(T_1), \forall_1(T_2))$, which will be accepted by $\mathcal{A}_\mathcal{C}$. Note that the branching between fair and unfair paths have been moved from the $\forall$-node to the $\exists$-node here.

(III) A fair non-terminating execution sequence has been removed by an under-approximation. For example, consider the expression:

$$\texttt{let}\ x : \textbf{int}[\lambda x.0 \leq x < y] = *_{\text{int}}\ \texttt{in}\ \texttt{if}\ 0 \leq x < y\ \texttt{then}\ fair_nonterminating()\ \texttt{else}\ unfair_nonterminating().$$

Here, the random number x is abstracted according to the predicate $\lambda x.0 \leq x < y$. If we do not have enough information to conclude that there exists x such that $0 \leq x < y$, then the expression is

abstracted to:

$$\exists\{\texttt{true} \to \texttt{let}\ x = \texttt{false}\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ M_1\ \texttt{else}\ M_2\},$$

which would generate $\exists_1(T_2)$ where T_2 contains only an unfair infinite path. In this case, we need to add a predicate on y needed to decide whether there exists x such that $0 \le x < y$.

Kuwahara et al. [22] made a similar classification on the first and second cases (where they needed to consider only terminating paths as error paths), but overlooked the third case.

The first and second cases (I) and (II) above can be detected by looking at each path of a counterexample tree. In the first case, T_2 in the example occurs as an *infeasible* (i.e., having no corresponding execution in the source program) unfair or terminating path, and in the second case, T_2 in the example occurs as a *feasible* unfair or terminating path. It is hard to detect the third case (III) from the counterexample tree. We can, however, detect such a possibility during the abstraction; in the rule PA-RAND in Figure 7, we can apply the quantifier elimination to $\exists x.b_1 Q_1(x) \wedge \cdots \wedge b_k Q_k(x)$, and add the resulting predicate to the abstraction type environment Γ. In the example above, the required predicate $0 < y$ is indeed obtained by the quantifier elimination of $\exists x.0 \le x < y$.

For dealing with cases (I) and (II), we pick each path of the counterexample tree, and check whether a finite prefix of the path of a certain length is feasible or not. If it is infeasible, we can conclude that it belongs to case (I). In that case, we can apply the standard technique [19] to find predicates to exclude out the infeasible path. If the finite prefix is feasible, both cases are possible. We, however, tentatively assume that (II) is the case, and find predicates to avoid the merging of paths; for this purpose, we can apply Kuwahara et al.'s technique for "Type II paths" [22]. During the CEGAR cycle (consisting of Steps 1–3), we gradually increase the length of the prefix. Although each path of a counterexample tree may be infinite, in all the cases (I)-(III), the depth of each problematic $\exists$- or $\forall$-node is finite, so that we can eventually eliminate it.

Example 4. Recall the program P'_3 in Example 1 and the counterexample in Section 4.2. If we pick the path $(\exists_2\forall_2 A\forall_1)^\omega$ then the path is feasible but unfair. This path occurs because there is no information for deciding whether the condition $0 \le r$ in the body of f when $b_{0<r} = \texttt{false}$. In order to split the case $0 \le r$, we pick a finite prefix of the path, e.g., $\exists_2\forall_2 A\forall_1$. We find a sufficient condition $R(r)$ on the value of r in the body of *rand*, in order for P'_3 to have a reduction sequence not corresponding to $\exists_2\forall_2 A\forall_1$. The constraint on R is expressed by:

$$\forall r.(R(r) \Rightarrow 0 \le r) \wedge \exists r.R(r).$$

Here, the part $\forall r.(R(r) \Rightarrow 0 \le r)$ expresses a sufficient condition for the first branch of the body of function f to be chosen, and the part $\exists r.R(r)$ expresses the satisfiability of $R(r)$. We can obtain $R = \lambda r.0 \le r$ as a solution of the constraint, and add it to the type of r in the body of $rand$. □

Remark 1. If a given program is actually fair terminating, either the whole CEGAR cycle continues indefinitely, finding new (but useless) predicates repeatedly, or Kuwahara et al.'s procedure for "Type II paths" get stuck. The latter case happens, for example, for the following program P, which is obviously fair-terminating under $\{(\texttt{Never}, A)\}$:

$$\textbf{main}\ () = \textbf{event}\ A;\ \textbf{main}().$$

In this case, the abstract program D_P generates the infinite tree $A(A(A(\cdots)))$, consisting of a single path, which itself is the counterexample generated by a model checker. The path is feasible (so it is not Type I), but since no two paths have been merged by the abstraction, the procedure for Type II path gets stuck, failing to find predicates to split the path. In such a case, our procedure in Step 3 runs forever, looking for a longer path for which Type I or Type II procedure is applicable.

6. Implementation and Experiments

This section reports an implementation and experimental results.

6.1 Implementation

We have implemented a tool for disproving fair termination as an extension of MoCHi [19]. It takes a program of a subset of OCaml, annotated with events and a fairness constraint, and tries to prove that the program is not fair terminating under the fairness constraint. Currently, the tool supports only singleton fairness constraints (i.e., those of the form $\{(A, B)\}$). As a higher-order model checker, we have used `HorSatP` [9] and modified it to enable counterexample generation. We used Z3 [8] as the back-end SMT solver used for predicate abstraction and predicate discovery.

6.2 Experiments

We ran our tool against several programs, all of which are fair non-terminating. The benchmark programs are written in an extension of the language in Section 2, where general let-expressions ($\texttt{let}\ x = e_1\ \texttt{in}\ e_2$), sequential compositions ($e_1; e_2$), etc. are allowed. The experiments were conducted on a machine with Intel Xeon E5-2680 v3 (2.50GHz, 16GB of memory) with timeout of 300 seconds.

Table 1 shows the result of the experiments. The column "program" shows the name of each program. The column "order" shows the largest order of functions in the program. The column "cycle" shows the number of CEGAR cycles. The columns "step 1", "step 2", and "step 3" show the times spent for each step, all measured in seconds. The column "total" shows the total running time.

The programs of name `xx-buggy` are variants of the benchmark programs in [24]; the original programs are fair terminating, and we have modified them to introduce fair non-termination. We briefly explain some of the other programs below. All the benchmark programs are available at `http://www-kb.is.s.u-tokyo.ac.jp/~watanabe/fair_nonterm/`.

The program `loop` is a simple non-terminating program consisting of a random integer generator, a conditional branch and events A and B. The program `update-max-CPS` is the following program:

$$\begin{aligned}
ev_a\ k &= \textbf{event}\ A;\ k\ ()\\
ev_b\ k &= \textbf{event}\ B;\ k\ ()\\
cont\ x\ () &= \texttt{let}\ y = *_{\text{int}}\ \texttt{in}\\
&\qquad \texttt{if}\ x < y\ \texttt{then}\ (f\ ev_b\ y)\ \texttt{else}\ (f\ ev_a\ x)\\
f\ ev\ x &= ev\ (cont\ x)\\
\textbf{main} &= \texttt{let}\ r = *_{\text{int}}\ \texttt{in}\ f\ ev_a\ r.
\end{aligned}$$

The variable x in this program represents the largest value generated by the random integer generator $*_{\text{int}}$ so far. The event B occurs only when the value of x is updated. Therefore, the program is fair terminating if and only if the set of integers generated by $*_{\text{int}}$ can generates has no upper bound. Since we haven't embedded any fairness assumption on $*_{\text{int}}$ into the program, the program is not fair terminating.

The program `call-twice` is:

$$\begin{aligned}
call_twice\ g &= g\ (); g\ ()\\
f\ () &= \texttt{let}\ x = *_{\text{int}}\ \texttt{in}\\
&\qquad \texttt{if}\ x < 0\ \texttt{then}\ (\textbf{event}\ B; ())\ \texttt{else}\\
&\qquad (\textbf{event}\ A; call_twice\ f)\\
\textbf{main} &= f\ ().
\end{aligned}$$

program	order	cycle	time [sec]			
			step 1	step 2	step 3	total
`intro`	1	2	0.18	0.16	0.12	0.53
`loop`	2	2	0.42	0.09	0.43	1.03
`loop-CPS`	3	2	0.65	0.21	0.39	1.36
`nested-if`	2	2	0.55	0.20	4.84	5.71
`op-loop`	2	2	0.81	0.20	0.47	1.60
`update-max`	1	3	1.53	5.18	1.15	8.04
`update-max-CPS`	3	2	0.70	0.21	1.24	2.27
`call-twice`	2	2	0.42	0.19	0.20	0.89
`odd-nonterm`	1	≥4	-	-	-	timeout
`odd-nonterm-annot`	1	1	0.87	4.31	-	5.24
`compose`	2	2	0.33	0.10	0.15	0.66
`murase-repeat-buggy`	2	2	0.62	0.12	0.94	1.79
`murase-closure-buggy`	2	2	0.36	0.05	0.14	0.63
`koskinen-1-buggy`	2	3	1.59	0.72	0.61	3.08
`koskinen-2-buggy`	1	5	4.50	21.79	1.97	28.58
`koskinen-3-1-buggy`	1	3	0.95	0.42	0.72	2.25
`koskinen-3-2-buggy`	1	≥3	-	-	-	timeout
`koskinen-3-2-buggy-annot`	1	1	0.46	0.90	-	1.48
`koskinen-3-3-buggy`	1	4	1.76	1.10	1.29	4.52

Table 1: Experimental results.

The function f first generates a random number x. If $x > 0$, the function raises an event B and terminates; otherwise, it raises an event A and calls itself twice. If $*_{\text{int}}$ generates positive and negative integers alternately, the execution is fair but non-terminating.

Our tool could disprove fair termination of the benchmark programs, except `odd-nonterm` and `koskinen-3-2-buggy`. The time-outs for `odd-nonterm` and `koskinen-3-2-buggy` are due to the failure to discover appropriate predicates (to be used for predicate abstraction). To confirm this point, we have prepared annotated versions of `odd-nonterm` and `koskinen-3-2-buggy` (`odd-nonterm-annot` and `koskinen-3-2-buggy-annot`), where predicates are given as hints. Our tool could successfully verify them. For `update-max` and `koskinen-2-buggy`, the major bottleneck is Step 2 for higher-order model checking. We expect that our tool works for larger programs if we improve the APT higher-order model checker. For `nested-if`, the time spent step 3 is also large. This is probably because, for this program, it was required to handle both of the cases (I) and (II) in Step 3.

7. Related Work

The (automated or non-automated) methods for proving temporal properties of higher-order programs have been studied recently, but to our knowledge, ours is the first method that can be used for disproving general[8] linear-time temporal properties of higher-order programs. Skalka et al. [31], Koskinen and Terauchi [20], and Hofmann and Chen [14] proposed type-based methods for proving temporal properties, and Murase et al. [24] have proposed a fully automated method for proving fair termination. None of those methods can be used for *disproving* temporal properties. Lester et al. [23] have also applied APT higher-order model checking to automated verification of temporal properties. Their technique can be used also for disproving properties, but it is applicable only to finite-data functional programs (where only finite data domains are allowed).

As already mentioned, our method is an extension of the method for disproving termination of higher-order functional programs proposed by Kuwahara et al. [22]. The extension is non-trivial, however.

[8] We focused on fair termination, but as mentioned in Section 1, arbitrary ω-regular properties can be reduced to fair termination.

For imperative programs, several methods [2, 5, 6] have been proposed for automatically disproving fair termination. Atig et al. [2] proposed a method for disproving fair termination, whose target is multi-threaded programs written in a first-order imperative language. Cook et al. [5, 6] proposed methods for proving temporal properties expressed in fair CTL or CTL*. Since the negation of fair termination can be represented as a fair CTL formula or a CTL* formula, their methods can disprove fair termination. Those methods are, however, limited to first-order programs and it is not clear how they can be extended to deal with higher-order programs. In fact, Atig et al.'s technique focuses on detecting fair infinite execution sequences that are "ultimately periodic", but fair infinite execution sequences of higher-order programs tend not to be so. Cook et al.'s technique [5] requires that the control flow graph of a program is finite. One may think of using defunctionalization [7] to convert higher-order programs to first-order programs and then applying a technique for first-order programs. However, since the defunctionalization typically encodes function closures using recursive data structures, the verification of the resulting program would actually become harder.

To our knowledge, our work is the first serious application of APT higher-order model checking to verification of infinite-data higher-order programs [18, 28]; the previous applications of higher-order model checking (to infinite-data programs) [17, 19, 21, 22, 29, 33] used only a subclass of higher-order model checking called trivial automata model checking. There were a few implementations of APT higher-order model checkers [9, 10, 25], but the implementation of counterexample generation is new.

8. Conclusion

We have proposed an automated method for disproving fair termination of higher-order functional programs. Our method is a non-trivial extension of Kuwahara et al.'s method for disproving plain termination [22]. Improving the efficiency of our implementation is left for future work; to that end, a more efficient APT higher-order model checker is required.

There are a number of limitations in our current verification method/implementation, to be addressed in future work. We list some of future work below.

- Tighter integration with fair termination verification [24]: As discussed in Remark 1, our procedure does not terminate when a given program is fair terminating; thus, one has to run our procedure for disproving fair termination and Murase et al.'s procedure for proving termination [24] must be executed in parallel. A tighter integration of the two procedures would be required.
- General temporal property verification: Implementing a tool for general temporal property verification is left for future work. As mentioned in Section 1, in theory, Vardi's technique can be used for reducing verification of arbitrary ω-regular properties to fair non-termination, but it is not clear whether the naive reduction yields a practical verification tool.
- Scalability: The current tool works for only small programs; to make the tool scalable, we need to improve the underlying higher-order model checker, predicate discovery engine, etc.
- Relative completeness: It is desirable that the method satisfies relative completeness in some sense, as in other verification methods for higher-order programs [24, 33]. It is left for future work to identify a reasonable condition and prove the relative completeness with respect to it.
- Extension of the target language: Since our method relies on higher-order model checking, dealing with recursively-typed or untyped programs is a fundamental challenge.

Acknowledgments

We would like to thank Hiroshi Unno for discussions, and anonymous referees for useful comments. This work was supported by JSPS Kakenhi 15H05706.

References

[1] K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Computer Science*, 3(3), 2007.

[2] M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting fair non-termination in multithreaded programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2012.

[3] A. Carayol and O. Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012*, pages 165–174. IEEE Computer Society, 2012.

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

[5] B. Cook, H. Khlaaf, and N. Piterman. On automation of CTL* verification for infinite-state systems. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2015.

[6] B. Cook, H. Khlaaf, and N. Piterman. Fairness for infinite-state systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2015.

[7] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, PPDP 2001*, pages 162–174. ACM, 2001.

[8] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[9] K. Fujima. HorSatP: A saturation-based higher-order model checker for APT, 2015. Tool available from the author.

[10] K. Fujima, S. Ito, and N. Kobayashi. Practical alternating parity tree automata model checking of higher-order recursion schemes. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013.

[11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

[12] E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[13] A. Haddad. Model checking and functional program transformations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013*, volume 24 of *LIPIcs*, pages 115–126. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[14] M. Hofmann and W. Chen. Abstract interpretation from büchi automata. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014*, pages 51:1–51:10. ACM, 2014.

[15] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2011.

[16] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, ICFP 2014*, pages 311–324. ACM, 2014.

[17] N. Kobayashi. Model checking higher-order programs. *J. ACM*, 60 (3):20, 2013.

[18] N. Kobayashi and C. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009*, pages 179–188. IEEE Computer Society, 2009.

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

[20] E. Koskinen and T. Terauchi. Local temporal reasoning. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014*, pages 59:1–59:10. ACM, 2014.

[21] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *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, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 392–411. Springer, 2014.

[22] T. Kuwahara, R. Sato, H. Unno, and N. Kobayashi. Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2015.

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

[24] A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. Temporal verification of higher-order functional programs. In *Proceedings of*

the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pages 57–68. ACM, 2016.

[25] R. P. Neatherway and C. L. Ong. TravMC2: higher-order model checking for alternating parity tree automata. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014*, pages 129–132. ACM, 2014.

[26] P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*, pages 446–456. ACM, 2015.

[27] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, ICFP 2014*, pages 139–152. ACM, 2014.

[28] C. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science, LICS 2006, Proceedings*, pages 81–90. IEEE Computer Society, 2006.

[29] C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 587–598. ACM, 2011.

[30] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI 2008*, pages 159–169. ACM, 2008.

[31] C. Skalka, S. F. Smith, and D. V. Horn. Types and trace effects of higher order programs. *J. Funct. Program.*, 18(2):179–249, 2008.

[32] T. Tsukada and C. L. Ong. Compositional higher-order model checking via ω-regular games over böhm trees. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014*, pages 78:1–78:10. ACM, 2014.

[33] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013*, pages 75–86. ACM, 2013.

[34] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Logic*, 51(1-2):79–98, 1991.

[35] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, ICFP 2014*, pages 269–282. ACM, 2014.

[36] 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*, pages 400–411. ACM, 2015.