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 transitive closure 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 verfication; 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

General Terms Algorithms, Languages, Theory, Verification

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 $\widetilde{w}$ and write $|\widetilde{w}|$ for the length of the sequence $\widetilde{w}$ 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 $Trans_P$ be the transition

[*] Author's current affiliation: Toshiba Solutions, Japan

relation of a program P (i.e., $(e, e') \in \mathit{Trans}_P$ 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 $(f\,\widetilde{w}, g\,\widetilde{v})$ where $f\,\widetilde{w}$ and $g\,\widetilde{v}$ are total applications that occur in an execution of P such that the call $g\,\widetilde{v}$ is made from $f\,\widetilde{w}$.[1] 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 *actual calls that happen in the program execution*, and not "static" call sites that appear in the program text.)

In principle, checking disjunctive well-foundedness of (the transitive closure of) the transition relation is sound and complete for termination even for higher-order functional programs (i.e., the method proposed by [6, 28] is language agnostic because any 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 (i.e., the call stack). For example, recall the definition of the Ackermann function:

$$\begin{array}{rl} \mathsf{ac}\,m\,n \;=\; & \mathtt{if}\ m=0\ \mathtt{then}\ n+1\\ & \mathtt{else\ if}\ n=0\ \mathtt{then}\ \mathsf{ac}\ (m-1)\ 1\\ & \mathtt{else}\ \mathsf{ac}\ (m-1)\ (\mathsf{ac}\ m\ (n-1)) \end{array}$$

Let P be a program that calls ac with some non-negative arguments. P is terminating, and there is a simple termination argument (i.e., disjunctively well-founded relation containing the transitive closure of the calling relation) sufficient to prove termination: $D_1 \cup D_2$ where

$$\begin{array}{l} D_1 = \{(\mathsf{ac}\,m\,n, \mathsf{ac}\,m'\,n') \mid m > m' \land m' \geq 0\}\\ D_2 = \{(\mathsf{ac}\,m\,n, \mathsf{ac}\,m'\,n') \mid n > n' \land n' \geq 0\} \end{array}$$

Indeed, $\mathit{Call}_P^+ \subseteq D_1 \cup D_2$ 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 $m, n > 0$.

$$\begin{array}{rl} \mathsf{ac}\,m\,n & \to\ \mathsf{ac}\ (m-1)\ (\mathsf{ac}\ m\ (n-1))\\ & \to\ \mathsf{ac}\ (m-1)\ (\mathsf{ac}\ (m-1)\ (\mathsf{ac}\ m\ (n-2)))\\ & \quad\vdots\\ & \to\ \mathsf{ac}\ (m-1)\ (\mathsf{ac}\ (m-1)\ (\ldots, (\mathsf{ac}\ m\ 0)\ldots))\\ & \to\ \mathsf{ac}\ (m-1)\ (\mathsf{ac}\ (m-1)\ (\ldots, (\mathsf{ac}\ (m-1)\ 1)\ldots))\\ & \quad\vdots \end{array}$$

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]

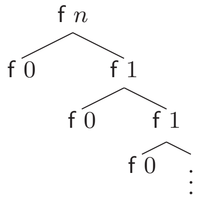

Figure 1. The graph of $\mathit{Call}_{P_{ex}}$.

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 π 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 π or q_i is true for infinitely many elements of π. 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 ϖ *fair* with respect to C if for each $(p_i, q_i) \in C$, either p_i is false for all elements of ϖ or q_i is true for some element of ϖ. For a binary relation R, let $R^{\mathit{fair}_C(+)} = \{(\varpi(1), \varpi(|\varpi|)) \mid \varpi \in \mathit{paths}(R) \land \varpi \text{ is fair wrt. } C\}$ where $\mathit{paths}(R) = \{\varpi \mid \forall i.1 \leq i < |\varpi| \Rightarrow (\varpi(i), \varpi(i+1)) \in R\}$. Intuitively, $R^{\mathit{fair}_C(+)}$ is 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 Pneuli et al. [26], Cook et al. [2] have shown that P is fair terminating under C if and only if $\mathit{Trans}_P^{\mathit{fair}_C(+)}$ (i.e., $R^{\mathit{fair}_C(+)}$ with $R = \mathit{Trans}_P$) 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 [2] 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 $\mathit{Call}_P^{\mathit{fair}_C(+)}$ is disjunctively well-founded. However, perhaps somewhat surprisingly, the approach turns out to be unsound.[3] To see this, consider the program P_{ex} that starts by calling the following function f with a positive argument:

$$\begin{array}{rl} \mathsf{f}\,x \;=\; & \mathtt{if}\ x \leq 0\ \mathtt{then}\ ()\\ & \mathtt{else}\ (\mathsf{f}\ 0); (\mathsf{f}\ 1) \end{array}$$

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

[1] We assume that the program is lambda-lifted [16] so that all function definitions occur (mutually recursively) at the top level.

[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 ω-regular temporal properties of program states. A similar issue applies in that setting.

f with the argument 0. Nonetheless, $\mathit{Call}_{P_{ex}}^{\mathit{fair}_C(+)}$ is disjunctively well-founded. Indeed, it can be seen from Figure 1 which shows the graph of $\mathit{Call}_{P_{ex}}$ (i.e., P_{ex}'s *call tree* [22]) that any infinite path in $\mathit{Call}_{P_{ex}}$ 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:

$$\begin{aligned}
&\mathit{Trans}_P^{+} = \{(e, e') \mid e \in \mathcal{R}_P \wedge e \rightarrow_P^{+} e'\}\\
&\mathit{Trans}_P^{\mathit{fair}_C(+)} = \{(e, e') \mid e \in \mathcal{R}_P \wedge e \rightarrow_P^{\mathit{fair}_C(+)} e'\}\\
&\mathit{Call}_P^{+} = \{(f\,\widetilde{w}, g\,\widetilde{v}) \mid E[f\,\widetilde{w}] \in \mathcal{R}_P \wedge f\,\widetilde{w} \rightarrow_P^{+} E'[g\,\widetilde{v}]\\
&\qquad\qquad\qquad\qquad \wedge \mathtt{arity}(f) = |\widetilde{w}| \wedge \mathtt{arity}(g) = |\widetilde{v}|\}
\end{aligned}$$

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

$$\begin{aligned}
\triangleright_P^C = \{(f\,\widetilde{w}, g\,\widetilde{v}) \mid E[f\,\widetilde{w}] \in \mathcal{R}_P \wedge f\,\widetilde{w} \rightarrow_P^{\mathit{fair}_C(+)} E'[g\,\widetilde{v}]\\
\wedge \mathtt{arity}(f) = |\widetilde{w}| \wedge \mathtt{arity}(g) = |\widetilde{v}|\}
\end{aligned}$$

That is, $\triangleright_P^C$ is the above characterization of Call_P^{+} but with $\rightarrow_P^{+}$ replaced by $\rightarrow_P^{\mathit{fair}_C(+)}$. 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 Call_P^{+} [22], an important advantage of this approach is that it avoids the explicit reasoning about changes in the calling context.

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

$$f\,n \triangleright_{P_{ex}}^C f\,1 \triangleright_{P_{ex}}^C f\,1 \triangleright_{P_{ex}}^C \dots$$

This follows because $f\,1 \rightarrow_{P_{ex}}^{+} (f\,0); (f\,1)$ and $(f\,0); (f\,1) \rightarrow_{P_{ex}}^{+} f\,1$, and therefore $f\,1 \rightarrow_{P_{ex}}^{\mathit{fair}_C(+)} f\,1$.

As a fair terminating example, consider P_{ex} again, but this time with respect to the fairness constraint $C' = \{(\mathsf{f}\,0, \mathsf{false})\}$. That is, f is called with 0 only finitely often in a trace fair with respect to C'. P_{ex} is fair terminating under C', because any infinite execution of P_{ex} is unfair, i.e., it contains infinitely many $\mathsf{f}\,0$ calls. We can show that $\triangleright_{P_{ex}}^{C'}$ is disjunctively well-founded. In fact, $\triangleright_{P_{ex}}^{C'} = \emptyset$ because $f\,n \not\rightarrow_{P_{ex}}^{\mathit{fair}_{C'}(+)} f\,i$ and $f\,1 \not\rightarrow_{P_{ex}}^{\mathit{fair}_{C'}(+)} f\,i$ 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 $f_1, \dots, f_n$ be the functions defined in P. Then, $\triangleright_P^C$ is disjunctively well-founded if and only if for each f_i, $\mathtt{REC}_{P,C}(f_i) = \{(\widetilde{w}, \widetilde{v}) \mid f_i\,\widetilde{w} \triangleright_P^C f_i\,\widetilde{v}\}$ 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 $\mathtt{REC}_{P,C}(f_i) \subseteq D$ 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 $\lceil P \rceil_{f,D,C}$ that is assertion safe if and only if $\mathtt{REC}_{P,C}(f) \subseteq D$ (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.

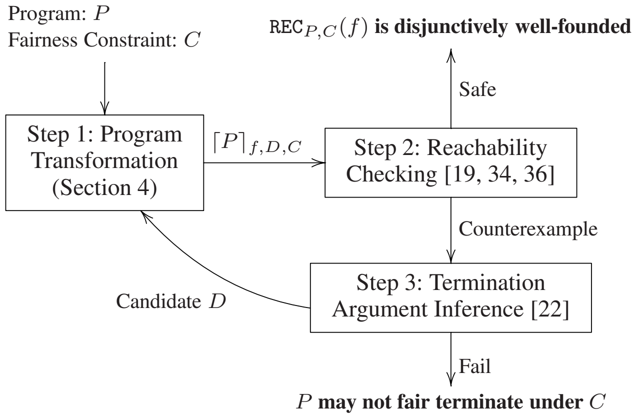
Program: P
Fairness Constraint: C
$\mathtt{REC}_{P,C}(f)$ is disjunctively well-founded
Safe
Step 1: Program Transformation (Section 4)
$\lceil P \rceil_{f,D,C}$
Step 2: Reachability Checking [19, 34, 36]
Counterexample
Step 3: Termination Argument Inference [22]
Candidate D
Fail
P may not fair terminate under C

Figure 2. Overview of the verification process.

In summary, our contributions are: (i) The first sound and complete approach to verification of arbitrary ω-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 [2] 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 $\lceil P \rceil_{f,D,C}$ from the initial state to an assertion error. That is, it is a finite sequence of (symbolic) executions ϖ of $\lceil P \rceil_{f,D,C}$ such that $\{(\widetilde{w}, \widetilde{v}) \mid [\![\varpi]\!](\widetilde{w}, \widetilde{v})\} \not\subseteq D$ where

$[\![\varpi]\!]$ is the strongest post condition of ϖ. (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

$$\{(\widetilde{w}, \widetilde{v}) \mid [\![\varpi]\!](\widetilde{w}, \widetilde{v})\} \subseteq \mathtt{REC}_{P,C}(f) \not\subseteq D$$

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 $\{(\widetilde{w}, \widetilde{v}) \mid [\![\varpi]\!](\widetilde{w}, \widetilde{v})\} \subseteq D'$. 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 *dis*proving 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

$$f\,1 \to_{P_{ex}}^{\mathit{fair}_C(+)} f\,1 \to_{P_{ex}}^{\mathit{fair}_C(+)} f\,1 \to_{P_{ex}}^{\mathit{fair}_C(+)} \cdots,$$

because in this case, $\mathtt{REC}_{P,C}(f)$ contains $(1, 1)$. Our method cannot, however, detect a non-looping non-termination, like

$$f\,1 \to_{P_{ex}}^{\mathit{fair}_C(+)} f\,2 \to_{P_{ex}}^{\mathit{fair}_C(+)} f\,3 \to_{P_{ex}}^{\mathit{fair}_C(+)} \cdots.$$

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 $\rhd_P^C$ (i.e., checking $\mathtt{REC}_{P,C}(f)$ is disjunctively well-founded for every f in P) , and Section 4 describes the program transformation that reduces the problem of checking if $\mathtt{REC}_{P,C}(f)$ 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.

$$\begin{aligned} e &::= v \mid \mathtt{event}\ A \mid \mathtt{rand}() \mid v_1\ \mathbf{op}\ v_2 \mid v_1 v_2 \\ &\quad\mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\ v &::= c \mid x \mid \langle x\ v_1\ \cdots\ v_k \rangle \end{aligned}$$

Figure 3. Source language: syntax

1.2 Related Work

To our knowledge, this paper is the first to propose a sound and complete approach to automatically verifying arbitrary ω-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 ω-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 Skalka et al.'s 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 ω-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 ω-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 $\star$), $\mathbf{op}$ ranges over the set of binary operators, and x over the set of variables. We write `true` and `false` for 1 and 0 respectively. We write $[\![\mathbf{op}]\!]$ for the semantics of the operator $\mathbf{op}$. The value $\langle x\ v_1\ \cdots\ v_k \rangle$ expresses a function closure; it occurs only at run-time. We write $[v_1/x_1, \ldots, v_i/x_i]e$ for the capture-avoiding parallel substitution of $v_1, \ldots, v_i$ for $x_1, \ldots, x_i$ in e.

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

[4] This occurs either when $\mathtt{REC}_{P,C}(f) \ni (\widetilde{v}, \widetilde{v})$ for some $\widetilde{v}$, or when a disjunctively well-founded relation D' does exist but is not found due to the incompleteness of the ranking function inference. In the former case, we can actually conclude that P does not fair terminate.

$$E[\texttt{event}\ A] \xrightarrow{A}_P E[\star] \qquad \text{(E-EV)}$$

$$\frac{n:\ \text{an integer}}{E[\texttt{rand}()] \xrightarrow{\epsilon}_P E[n]} \qquad \text{(E-RAND)}$$

$$\frac{v_1\ [\![\mathbf{op}]\!]\ v_2 = v}{E[v_1\ \mathbf{op}\ v_2] \xrightarrow{\epsilon}_P E[v]} \qquad \text{(E-OP)}$$

$$E[f] \xrightarrow{\epsilon}_P E[\langle f \rangle] \qquad \text{(E-CLOS0)}$$

$$\frac{i < \texttt{arity}(f)}{E[\langle f\, v_1\ \cdots\ v_{i-1}\rangle v_i] \xrightarrow{\epsilon}_P E[\langle f\, v_1\ \cdots\ v_i\rangle]} \qquad \text{(E-CLOS1)}$$

$$\frac{i = \texttt{arity}(f) \qquad f\, x_1\ \cdots\ x_i = e \in D}{E[\langle f\, v_1\ \cdots\ v_{i-1}\rangle v_i] \xrightarrow{\epsilon}_P E[[v_1/x_1,\ldots,v_i/x_i]e]} \qquad \text{(E-CALL)}$$

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

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

$$E[\texttt{let}\ x = v\ \texttt{in}\ e] \xrightarrow{\epsilon}_P E[[v/x]e] \qquad \text{(E-LET)}$$

Figure 4. Source language: reduction relation

A *program* P is a set of function definitions of the form:

$$f\ x_1\ \cdots\ x_k = e.$$

We write $\texttt{arity}(f)$ for the number of formal parameters, i.e., k. We assume that P contains the definition of the special "main" function `main` that takes a unit value and (if terminates) returns a unit value.

The set of *evaluation contexts*, ranged over by E, is given by:

$$E ::= [\,]\ |\ \texttt{let}\ x = E\ \texttt{in}\ e.$$

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

We write $\xrightarrow{\ell_1\cdots\ell_n}{}^{*}_{P}$ for the relational composition $\xrightarrow{\ell_1}_P \cdots \xrightarrow{\ell_n}_P$. Let $\sigma = \ell_1\ell_2\ell_3\cdots$ be a possibly infinite sequence of labels. We write $e \xrightarrow{\sigma}{}^{*}_{P}$ if there exists a (possibly infinite) sequence of expressions $e_1, e_2, \ldots$ such that

$$e \xrightarrow{\ell_1}_P e_1 \xrightarrow{\ell_2}_P e_2 \xrightarrow{\ell_3}_P \cdots$$

with $\sigma = \ell_1\ell_2\ell_3\cdots$.

Example 2.1 The example on the function f in Section 1 is expressed as the following program P_{ex}:

$$\begin{array}{l} f\ x = \texttt{if}\ x < 0\ \texttt{then}\ \star\ \texttt{else} \\ \qquad\qquad \texttt{if}\ x = 0\ \texttt{then}\ \texttt{event}\ A\ \texttt{else}\ (f\ 0); (f\ 1) \\ \texttt{main}\ y = \texttt{let}\ x = \texttt{rand}()\ \texttt{in}\ f\ x \end{array}$$

Here, we have inserted $\texttt{event}\ A$ to capture the event that $f\ 0$ is called. □

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

$$\begin{array}{l} \mathit{repeat}\ g = \texttt{let}\ x = \texttt{rand}()\ \texttt{in}\ g\ x; \mathit{repeat}\ g \\ f\ x = \texttt{if}\ x > 0\ \texttt{then}\ f(x-1)\ \texttt{else}\ \texttt{event}\ A \\ \texttt{main}\ y = \mathit{repeat}\ f \end{array}$$

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 σ be a possibly infinite sequence of events. We write $\sigma \models C$ when, for every $i \in \{1, \ldots, n\}$, if A_i occurs infinitely often in σ, 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

$$\texttt{main}\ \star \xrightarrow{\ell_1}_P e_1 \xrightarrow{\ell_2}_P e_2 \xrightarrow{\ell_3}_P \cdots,$$

$\ell_1\ell_2\ell_3\cdots \models C$ 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 ω-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 `event Call`, so that every infinite reduction sequence contains infinitely may occurrences of the event `Call`. Assume also that a special event `Never` never occurs in programs.

- Let C be $\{(A, \texttt{Never})\}$. Then a program P is fair terminating under C if and only if A occurs infinitely often in any non-terminating run of P. Indeed, if P is fair terminating, then the fairness constraint C must be violated by any infinite run of the program, which implies that A must occur infinitely often. To see the converse, we consider the contraposition. Suppose that the program is *not* fair terminating, i.e., there is a fair, infinite reduction sequence $\texttt{main}\star \xrightarrow{\sigma}{}^{*}_{P}$. Then $\sigma \models C$, which implies that either (i) A occurs only finitely often or (ii) both A and `Never` occurs infinitely often. The latter cannot be the case by the assumption on `Never`; hence A occurs only finitely often.
- Let C be $\{(A, \texttt{Never}), (\texttt{Call}, B)\}$. Then a program P is fair terminating under C if and only if, for every infinite run of P, if B occurs infinitely often, so does A. To see why, notice that C is violated just if either $(A, \texttt{Never})$ or $(\texttt{Call}, B)$ is violated. Since `Never` never happens and `Call` occurs infinitely often, $(A, \texttt{Never})$ being violated means that A occurs infinitely often, and $(\texttt{Call}, B)$ being violated means that B occurs only finitely often.
- Let P' be the program obtained from a program P by replacing `event Call` with:

  $$\texttt{if}\ \mathit{flag}\ \texttt{then event Call}_A\ \texttt{else event Call}.$$

  Here, *flag* is a state variable (which is added as an additional parameter of each function) that expresses whether A has happened before. Then, P' is fair terminating under $\{(\texttt{Call}_A, \texttt{Never})\}$ if and only if A occurs eventually in every infinite run of P.

- Let P' be the program obtained from a program P by replacing the definition of the main function: $\mathtt{main}\,x = e$ with

  $$\begin{array}{l}\mathtt{main}\,x = e; \mathit{loop}\,\star \\ \mathit{loop}\,x = \mathtt{event\ Never}; \mathit{loop}\,x\end{array}$$

  Then P' is fair terminating under $\{(\mathtt{Call}, \mathtt{Never})\}$ if and only if P never terminates.

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

Note that the use of Streett fairness constraints in our work stipulates that $A_{\neg\phi}$ is represented by a Streett automaton. Streett automata are preferred over Büchi automata in our setting because a non-deterministic Büchi automaton is required to express an ω-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 relation* (cf. Section 1.1).

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

Definition 3.2 Let P be a program, f a function defined in P, and C a fairness constraint. We write $f\,\widetilde{v} \rhd_P^C g\,\widetilde{w}$ if there exist $\sigma_1, \sigma_2, E_1, E_2$ such that: (i) $\mathtt{main}\ \star \xrightarrow{\sigma_1}{}^{*}_{P} E_1[f\,\widetilde{w}]$, (ii) $f\,\widetilde{w} \xrightarrow{\sigma_2}{}^{+}_{P} E_2[g\,\widetilde{v}]$, (iii) $\sigma_2 \models_{\mathtt{fin}} C$, and (iv) $\mathtt{arity}(f) = |\widetilde{w}|$ and $\mathtt{arity}(g) = |\widetilde{v}|$. We write $\mathtt{REC}_{P,C}(f)$ for the set

$$\{(\widetilde{w}, \widetilde{v}) \mid f\,\widetilde{w} \rhd_P^C f\,\widetilde{v}\},$$

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, $\mathtt{REC}_{P,C}(f)$ is well-founded (i.e., there is no infinite chain $\widetilde{v}_0\,\mathtt{REC}_{P,C}(f)\,\widetilde{v}_1 \mathtt{REC}_{P,C}(f)\,\widetilde{v}_2\,\mathtt{REC}_{P,C}(f)\,\widetilde{v}_3 \cdots$) for every function f defined in P.

Since the relation $\mathtt{REC}_{P,C}(f)$ is transitive, $\mathtt{REC}_{P,C}(f)$ is well-founded if and only if $\mathtt{REC}_{P,C}(f)$ 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 $\mathtt{REC}_{P,C}(f) \subseteq D_f$. We call the problem of proving $\mathtt{REC}_{P,C}(f) \subseteq D_f$ 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_{rep} raises events A infinitely often, it suffices to verify

$$\begin{array}{ll}\mathtt{REC}_{P,C}(f) \subseteq D_f & \mathtt{REC}_{P,C}(\mathit{repeat}) \subseteq D_{\mathit{repeat}} \\ \mathtt{REC}_{P,C}(\mathtt{main}) \subseteq D_{\mathtt{main}} & \end{array}$$

for $C = \{(A, \mathtt{Never})\}$, $D_f = \{(m, n) \mid m > n \geq 0\}$, and $D_{\mathit{repeat}} = D_{\mathtt{main}} = \emptyset$. Note that $\mathtt{REC}_{P,C}(\mathit{repeat}) = \emptyset$ because, whenever $\mathit{repeat}\ f \xrightarrow{\sigma}{}^{*} E[\mathit{repeat}\ f]$, σ contains A, so that $\sigma \not\models C$. □

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 $\mathtt{REC}_{P,C}(f)$ is well-founded for every function f in P, then P is fair terminating.

Proof The proof is by contradiction. Suppose that $\mathtt{REC}_{P,C}(f)$ is well-founded for every function f in P, but P is not fair terminating, i.e., there exists an infinite reduction sequence π that is fair. Since π is an infinite sequence, there must be a function f such that π can be decomposed to:

$$\mathit{main} \xrightarrow{\sigma_0}{}^{*}_{P} e_1 \xrightarrow{\sigma_1}{}^{*}_{P} e_2 \xrightarrow{\sigma_2}{}^{*}_{P} \cdots$$

where $e_i = E_1[\cdots E_i[f\widetilde{v}_i] \cdots]$ and $f\widetilde{v}_i \xrightarrow{\sigma_i}{}^{*}_{P} E_{i+1}[f\widetilde{v}_{i+1}]$ [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 π. Let P_i and Q_i be $\{A \mid (A, B) \in C_i\}$ and $\{B \mid (A, B) \in C_i\}$ respectively. As π is fair, there exists k such that, in $\sigma_k \sigma_{k+1} \cdots$, no event in P_1 occurs and every event in Q_2 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_2 occurs in $\sigma_{n_i} \sigma_{n_i+1} \cdots \sigma_{j-1}$. Then, we have an infinite sequence:

$$f\widetilde{v}_{n_1} \rhd_P^C f\widetilde{v}_{n_2} \rhd_P^C f\widetilde{v}_{n_3} \rhd_P^C \cdots.$$

This contradicts the assumption that $\mathtt{REC}_{P,C}(f)$ is well-founded. □

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

Proof The proof is by contradiction. Suppose that $\mathtt{REC}_{P,C}(f)$ is not well-founded for some f. Then there must be an infinite sequence

$$f\widetilde{v}_1 \rhd_P^C f\widetilde{v}_2 \rhd_P^C f\widetilde{v}_3 \rhd_P^C \cdots.$$

By the definition of $\rhd_P^C$, there exist $E_1, E_2, \ldots$ where $\mathit{main}\,\star \xrightarrow{\sigma_0}{}^{*}_{P} E_1[f\widetilde{v}_1]$ and $f\widetilde{v}_i \xrightarrow{\sigma_i}{}^{*}_{P} E_{i+1}[f\widetilde{v}_{i+1}]$ with $\sigma_i \models_{\mathtt{fin}} C$ for every $i \geq 1$. Thus, we have an infinite reduction sequence π:

$$\mathit{main}\,\star \xrightarrow{\sigma_0}{}^{*}_{P} E_1[f\widetilde{v}_1] \xrightarrow{\sigma_1}{}^{*}_{P} E_1[E_2[f\widetilde{v}_2]] \xrightarrow{\sigma_2}{}^{*}_{P},$$

where for each σ_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 π. Then, there exist infinitely many i's such that A occurs in σ_i. But then B also occurs for each of such i's. Thus, B also occurs infinitely often. This implies that π is an infinite fair reduction sequence, hence a contradiction. □

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 π be a possibly infinite reduction sequence from $\texttt{main}\,\star$. Let us write $\pi(i)$ for the ith element of π. That is,

$$\texttt{main}\,\star\ = \pi(1) \xrightarrow{\ell_1}_P \pi(2) \xrightarrow{\ell_2}_P \cdots$$

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

As in [22], to check $\texttt{REC}_{P,C}(f) \subseteq D_f$, the arguments $\widetilde{w}$ 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 $\widetilde{v}$, we assert that $D_f(\widetilde{w}, \widetilde{v})$ holds. A further twist is required, however. We also need to keep information about the set σ 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 $\widetilde{v}$, we assert $D_f(\widetilde{w}, \widetilde{v})$ only if $\sigma \models_{\texttt{fin}} C$ holds.

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 $\{(A, \texttt{Never})\}$. To check that $\texttt{REC}_{P_{rep},C}(f) \subseteq D_f$, we transform P_{rep} to the following program.

$$\begin{array}{l}
\mathit{repeat}\ s\ w\ g = \texttt{let}\ x = \texttt{rand}()\ \texttt{in} \\
\qquad\qquad\quad \texttt{let}\ (s', \_) = g\ s\ w\ x\ \texttt{in}\ \mathit{repeat}\ s'\ w\ g \\
f\ s\ w\ x = \texttt{assert}(s = \texttt{false} \Rightarrow w > x \geq 0); \\
\qquad\quad\ \ \texttt{let}\ (s, w) = \\
\qquad\qquad\ \ \texttt{if}\ \texttt{rand}()\ \texttt{then}\ (\texttt{false}, x)\ \texttt{else}\ (s, w)\ \texttt{in} \\
\qquad\quad\ \ \texttt{if}\ x > 0\ \texttt{then}\ f\ s\ w\ (x-1)\ \texttt{else}\ (\texttt{true}, \star) \\
\texttt{main}\ s\ w\ y = \mathit{repeat}\ s\ w\ f
\end{array}$$

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 $f\ \ w$. 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 function f, it is asserted that if an event A has not occurred, $w > x \geq 0$ (i.e., $D_f(w, x)$) must hold. If the assertion fails, the program is aborted. Otherwise, the pair (s, w) is non-deterministically kept or updated to $(\texttt{false}, x)$; 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 recursive call $f\ w \xrightarrow{\sigma}{}^{*}_{P_{rep}} E[f\ v]$ 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 $\sigma \models_{\texttt{fin}} C$. Therefore, $\texttt{REC}_{P_{rep},C}(f) \subseteq D_f$ if and only if the assertion in $\texttt{main false}\ \bot\ \star$ 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 repeat, the event information returned by $g\,s\,w\,x$ is bound to s' and fed to the subsequent recursive call $\mathit{repeat}\ s'\,w\,g$. 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 repeat are both w: $g\,s\,w\,x$ and $\mathit{repeat}\ s'\ w\ g$. 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.

$$\begin{array}{l}
e ::= \cdots \mid \texttt{fail} \mid \texttt{let}\ (x_1, \ldots, x_k) = e_1\ \texttt{in}\ e_2 \\
v ::= \cdots \mid (v_1, \ldots, v_k) \mid \bot
\end{array}$$

Here, $\texttt{fail}$ 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 $\texttt{assert}(e)$ for $\texttt{let}\ x = e\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ \star\ \texttt{else}\ \texttt{fail}$.

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

$$E ::= \cdots \mid \texttt{let}\ (x_1, \ldots, x_k) = E\ \texttt{in}\ e$$

$$\begin{array}{l}
E[\texttt{let}\ (x_1, \ldots, x_k) = (v_1, \ldots, v_k)\ \texttt{in}\ e] \\
\qquad\qquad \xrightarrow{\epsilon}_P E[[v_1/x_1, \ldots, v_k/x_k]e]
\end{array}$$

$$E[\texttt{fail}] \xrightarrow{\epsilon}_P \texttt{fail}$$

We remark that we have defined $\texttt{fail}$ 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 $P = \{f_1\ \widetilde{x}_1 = e_1, \ldots, f_n\ \widetilde{x}_n = e_n\}$ is transformed to:

$$\lceil P \rceil_{f,D,C} = \bigcup_{i \in \{1,\ldots,n\}} \lceil f_i\ \widetilde{x}_i = e_i \rceil_{f,D,C}.$$

Here, each function definition $g\,x_1\ \cdots\ x_k = e$ is translated to the set $\lceil g\,x_1\ \cdots\ x_k = e \rceil_{f,D,C}$ of function definitions, given by:

$$\begin{array}{l}
\lceil g\,x_1\ \cdots\ x_k = e \rceil_{f,D,C} = \\
\{\ g\ s\ w\ x_1 = (s, g^{(1)}\ x_1), \\
\quad g^{(1)}\ x_1\ s\ w\ x_2 = (s, g^{(2)}\ x_1\ x_2), \\
\quad \ldots, \\
\quad g^{(k-1)}\ x_1\ \cdots\ x_{k-1}\ s\ w\ x_k = e'\ \}
\end{array}$$

As in the transformation of P_{rep} 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,

$$\begin{array}{l}
\lceil v\rceil_{s,w,C} = (s, \lceil v\rceil_V)\\
\lceil \mathtt{rand}()\rceil_{s,w,C} = (s,\mathtt{rand}())\\
\lceil \mathtt{event}\ A\rceil_{s,w,C} = (s\{A := \mathtt{true}\}, \star)\\
\lceil v_1\ \mathbf{op}\ v_2\rceil_{s,w,C} = (s, v_1\ \mathbf{op}\ v_2)\\
\lceil v_1 v_2\rceil_{s,w,C} = \lceil v_1\rceil_V\ s\ w\ \lceil v_2\rceil_V\\
\lceil \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rceil_{s,w,C}\\
\qquad\qquad = \mathtt{if}\ v\ \mathtt{then}\ \lceil e_1\rceil_{s,w,C}\ \mathtt{else}\ \lceil e_2\rceil_{s,w,C}\\
\lceil \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\rceil_{s,w,C}\\
\qquad\qquad = \mathtt{let}\ (s', x) = \lceil e_1\rceil_{s,w,C}\ \mathtt{in}\ \lceil e_2\rceil_{s',w,C}\\
\lceil c\rceil_V = c \qquad \lceil x\rceil_V = x\\
\lceil\langle f\ v_1\ \cdots\ v_k\rangle\rceil_V = \langle f^{(k)}\ \lceil v_1\rceil_V\ \cdots\ \lceil v_k\rceil_V\rangle
\end{array}$$

Figure 5. The transformation of expressions and values.

called an *event history*, keeps information about the events that have occurred since the call $f\ w$. If $C = \{(A_1, B_1), \dots, (A_n, B_n)\}$, then s is a nested tuple of the form:

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

where p_i (q_i, resp.) is a Boolean that expresses whether A_i (B_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 $g^{(1)}, \dots, g^{(k-1)}$. The body e' of $g^{(k-1)}$ is:

$$\begin{array}{l}
\mathtt{assert}(\mathtt{fair}_s \Rightarrow D^{\#}(w_k, (x_1, \dots, x_k)));\\
\mathtt{let}\ (s, w) =\\
\qquad \mathtt{if}\ \mathtt{rand}()\ \mathtt{then}\ (s_{\mathtt{init}}, (x_1, \dots, x_k))\ \mathtt{else}\ (s, w)\ \mathtt{in}\\
\lceil e\rceil_{s,w,C}
\end{array}$$

if $g = f$, and $\lceil e\rceil_{s,w,C}$ otherwise (where the transformation $\lceil e\rceil_{s,w,C}$ for expressions is given later). Here, $\mathtt{fair}_s$ denotes the expression

$$\begin{array}{l}
\mathtt{let}\ ((p_1, q_1), \dots, (p_n, q_n)) = s\ \mathtt{in}\\
\qquad (p_1 \Rightarrow q_1)\ \&\&\ \cdots\ \&\&\ (p_n \Rightarrow q_n),
\end{array}$$

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

$$((\mathtt{false}, \mathtt{false}), \dots, (\mathtt{false}, \mathtt{false})).$$

$D^{\#}(w, v)$ is an expression such that $D^{\#}(w, v)$ evaluates to $\mathtt{true}$ if (i) $w = (\lceil w_1\rceil_V, \dots, \lceil w_k\rceil_V)$ and $v = (\lceil v_1\rceil_V, \dots, \lceil v_k\rceil_V)$ with $((w_1, \dots, w_k), (v_1, \dots, v_k)) \in D$ or (ii) $w = \bot$, and evaluates to $\mathtt{false}$ otherwise.[5] In the definition of e' above, s contains information about the sequence σ of events that have occurred since an ancestor call to f, and $\mathtt{fair}_s$ expresses whether $\sigma \models_{\mathtt{fin}} C$ holds.

The transformations of expressions and values, denoted by $\lceil\cdot\rceil_{s,w,C}$ and $\lceil\cdot\rceil_V$ 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, $s\{A := \mathtt{true}\}$ is an expression for updating the event history by replacing the elements corresponding to A with $\mathtt{true}$. More precisely, if $C = \{(A_{1,1}, A_{2,1}), \dots, (A_{1,n}, A_{2,n})\}$, then $s\{A := \mathtt{true}\}$ is:

$$\begin{array}{l}
\mathtt{let}\ ((p_{1,1}, p_{2,1}), \dots, (p_{1,n}, p_{2,n})) = s\ \mathtt{in}\\
\qquad ((p'_{1,1}, p'_{2,1}), \dots, (p'_{1,n}, p'_{2,n}))
\end{array}$$

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

In the following, we fix the fairness constraint C and omit it in the transformation notation (e.g., we write $\lceil P\rceil_{f,D}$ instead of $\lceil P\rceil_{f,D,C}$). The following theorems guarantee that our reduction from fair binary reachability to plain reachability is sound and complete.

Theorem 4.1 (Correctness of the transformation)
Suppose $\lceil P\rceil_{f,D} = P'$. Then, $\mathtt{REC}_{P,C}(f) \not\subseteq D$ if and only if $\mathtt{main}\, s_{\mathtt{init}} \perp \star \rightarrow^*_{P'} \mathtt{fail}$.

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 $\lceil P\rceil_{f,D} = P'$ and $\mathtt{REC}_{P,C}(f) \not\subseteq D$. Then, if $\mathtt{REC}_{P,C}(f) \not\subseteq D$ then $\mathtt{main}\, s_{\mathtt{init}} \perp \star \rightarrow^*_{P'} \mathtt{fail}$.

Theorem 4.3 (Completeness of the transformation)
Suppose $\lceil P\rceil_{f,D} = P'$. Then, if $\mathtt{main}\, s_{\mathtt{init}} \perp \star \rightarrow^*_{P'} \mathtt{fail}$ then $\mathtt{REC}_{P,C}(f) \not\subseteq D$.

4.2 Proof of Soundness (Theorem 4.2)

The soundness follows from the following properties: (i) each reduction of an expression e in the source program is simulated by the reductions of the corresponding expression $\lceil e\rceil_{s,w}$ 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:

$$\begin{array}{l}
\lceil[\,]\rceil_w = [\,]\\
\lceil \mathtt{let}\ x = E\ \mathtt{in}\ e\rceil_w = \mathtt{let}\ (s, x) = \lceil E\rceil_w\ \mathtt{in}\ \lceil e\rceil_{s,w}
\end{array}$$

Lemma 4.4 If E is a context for the source language, then $\lceil E\rceil_w$ is a context for the target language. Furthermore, $\lceil E[e]\rceil_{s,w} = \lceil E\rceil_w[\lceil e\rceil_{s,w}]$ holds.

Proof This follows by a straightforward induction on the structure of E. □

Lemma 4.5 $\lceil [v/x]e\rceil_{s,w} = [\lceil v\rceil_V/x]\lceil e\rceil_{s,w}$.

Proof This follows by a straightforward induction on the structure of e. □

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

$$\epsilon(s) = s \qquad A \cdot \sigma(s) = \sigma([\![s\{A := \mathtt{true}\}]\!])$$

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

Lemma 4.6 Let P' be $\lceil P\rceil_{f,D_0}$, If $e_1 \xrightarrow{\ell}_P e_2$, then $\lceil e_1\rceil_{s,w} \rightarrow^*_{P'} \lceil e_2\rceil_{\ell(s),w}$.

Proof The proof proceeds by a case analysis on the rule used for deriving $e_1 \xrightarrow{\ell}_P e_2$. We discuss only the main cases; the other cases are straightforward.

- Case E-Ev: In this case, $e_1 = E[\mathtt{event}\ A]$ and $e_2 = E[\star]$ with $\ell = A$. By Lemma 4.4 and the definition of the transformation, we have:

$$\begin{array}{l}
\lceil e_1\rceil_{s,w} = \lceil E\rceil_w[(s\{A := \mathtt{true}\}, \star)]\\
\lceil e_2\rceil_{\ell(s),w} = \lceil E\rceil_w[(\ell(s), \star)]
\end{array}$$

Thus, we have $\lceil e_1\rceil_{s,w} \rightarrow^*_{P'} \lceil e_2\rceil_{\ell(s),w}$ as required.

[5] 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.

- Case E-CLOS: In this case, $e_1 = E[\langle g\, v_1 \cdots v_{i-1}\rangle v_i]$ and $e_2 = E[\langle g\, v_1 \cdots v_i\rangle]$ with $\ell = \epsilon$ and $i < \mathtt{arity}(g)$. By Lemma 4.4 and the definition of the transformation, we have:

  $$\begin{array}{l}\lceil e_1\rceil_{s,w} = \lceil E\rceil_w[\langle g^{(i-1)}\lceil v_1\rceil_V \cdots \lceil v_{i-1}\rceil_V\rangle\ s\ w\ \lceil v_i\rceil_V]\\ \lceil e_2\rceil_{\ell(s),w} = \lceil E\rceil_w[(\ell(s), \langle g^{(i)}\lceil v_1\rceil_V \cdots \lceil v_i\rceil_V\rangle)]\end{array}$$

  By inspection of the reduction rules, we have

  $$\begin{array}{l}\lceil e_1\rceil_{s,w} \to_{P'} \lceil E\rceil_w[(s, g^{(i)}\lceil v_1\rceil_V \cdots \lceil v_i\rceil_V)]\\ \qquad \to^*_{P'} \lceil E\rceil_w[(s, \langle g^{(i)}\lceil v_1\rceil_V \cdots \lceil v_i\rceil_V\rangle)].\end{array}$$

  Since $\ell(s) = s$, we have $\lceil e_1\rceil_{s,w} \to_{P'} \lceil e_2\rceil_{\ell(s),w}$ as required.

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

  $$\begin{array}{l}\lceil e_1\rceil_{s,w} = \lceil E\rceil_w[\langle g^{(i-1)}\lceil v_1\rceil_V \cdots \lceil v_{i-1}\rceil_V\rangle s\ w\ \lceil v_i\rceil_V]\\ \to^+_{P'} \lceil E\rceil_w[[\lceil v_1\rceil_V/x_1, \ldots, \lceil v_i\rceil_V/x_i]\lceil e\rceil_{s,w}]\\ = \lceil E\rceil_w[\lceil[\lceil v_1\rceil_V/x_1, \ldots, \lceil v_i\rceil_V/x_i]e\rceil_{s,w}]\\ = \lceil e_2\rceil_{s,w} = \lceil e_2\rceil_{\ell(s),w}\end{array}$$

  by Lemma 4.5 and the definition of the transformation.

  If $g = f$, then we have

  $$\begin{array}{l}\lceil e_1\rceil_{s,w} = \lceil E\rceil_w[\langle g^{(i-1)}\lceil v_1\rceil_V \cdots \lceil v_{i-1}\rceil_V\rangle s\ w\ \lceil v_i\rceil_V]\\ \to^+_{P'} \lceil E\rceil_w[\mathtt{assert}(\mathtt{fair}_s \Rightarrow D_0^{\#}(w, \widetilde{\lceil v\rceil_V})); \cdots]\\ \to^+_{P'} \lceil E\rceil_w[\mathtt{let}\ (s, w) =\\ \qquad \mathtt{if\ rand()\ then}\ \cdots\ \mathtt{else}\ (s, w)\ \mathtt{in}\ [\widetilde{\lceil v\rceil_V}/\widetilde{x}]\lceil e\rceil_{s,w}]\\ \to^+_{P'} \lceil E\rceil_w[[\widetilde{\lceil v\rceil_V}/\widetilde{x}]\lceil e\rceil_{s,w}]\\ = \lceil e_2\rceil_{s,w} = \lceil e_2\rceil_{\ell(s),w},\end{array}$$

  as required. □

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

Lemma 4.7 Suppose $f\, x_1 \cdots x_k = e \in P$ and $\lceil P\rceil_{f,D_0} = P'$. Then,

$$\lceil\langle f\, v_1 \cdots v_{k-1}\rangle v_k\rceil_{s,w} \to^*_{P'} \lceil[v_1/x_1, \ldots, v_k/x_k]e\rceil_{s_{\mathtt{init}},w'}$$

with $w' = (\lceil v_1\rceil_V, \ldots, \lceil v_k\rceil_V)$.

Proof This follows immediately from the definition of the transformation and Lemma 4.5. □

Proof of Theorem 4.2 Let $\lceil P\rceil_{f,D} = P'$ and $f\ w_1\ \cdots\ w_k \rhd^C_P f\ v_1\ \cdots\ v_k$ with $((w_1, \ldots, w_k), (v_1, \ldots, v_k)) \notin D$. Let P'' be $\lceil P\rceil_{f,D_0} = P''$ By the definition of $f\ w_1\ \cdots\ w_k \rhd^C_P f\ v_1\ \cdots\ v_k$, we have

$$\begin{array}{l}\mathtt{main} \star \xrightarrow{\sigma_1}{}^*_P E_1[\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k]\\ \langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k \xrightarrow{\epsilon}_P [w_1/x_1, \ldots, w_k/x_k]e\\ \qquad\qquad\qquad \xrightarrow{\sigma_2}{}^*_P E_2[\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k]\\ f\ x_1\ \cdots\ x_k = e \in P\\ \sigma_2 \models_{\mathtt{fin}} C\end{array}$$

By Lemmas 4.6 and 4.7, we have:

$$\begin{array}{l}\mathtt{main}\ s_{\mathtt{init}} \perp \star\\ \to^*_{P''} \lceil E_1[\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k]\rceil_{\sigma_1(s_{\mathtt{init}}),\perp}\\ = E_1'[\lceil\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k\rceil_{\sigma_1(s_{\mathtt{init}}),\perp}]\\ \to^*_{P''} E_1'[\lceil[w_1/x_1, \ldots, w_k/x_k]e\rceil_{s_{\mathtt{init}},(w_1', \ldots, w_k')}]\\ \to^*_{P''} E_1'[\lceil E_2[\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k]\rceil_{\sigma_2(s_{\mathtt{init}}),(w_1', \ldots, w_k')}]\\ = E_1'[E_2'[\lceil\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k\rceil_{\sigma_2(s_{\mathtt{init}}),(w_1', \ldots, w_k')}]]\end{array}$$

where $w_i' = \lceil w_i\rceil_V$, $E_1' = \lceil E_1\rceil_\perp$, and $E_2' = \lceil E_2\rceil_{(w_1', \ldots, w_k')}$.

Thus, we have either $\mathtt{main}\ s_{\mathtt{init}} \perp \star \to^*_{P'} \mathtt{fail}$, or

$$\begin{array}{l}\mathtt{main}\ s_{\mathtt{init}} \perp \star\\ \qquad \to^*_{P'} E_1'[E_2'[\lceil\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k\rceil_{\sigma_2(s_{\mathtt{init}}),(w_1', \ldots, w_k')}]].\end{array}$$

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

$$\begin{array}{l}\lceil\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k\rceil_{\sigma_2(s_{\mathtt{init}}),(w_1', \ldots, w_k')}\\ = \langle f^{(k-1)}\ v_1'\ \cdots\ v_{k-1}'\rangle\ s_{\mathtt{init}}\ (w_1', \ldots, w_k')\ v_k'\\ \to^*_{P'} E_3[\mathtt{assert}(\mathtt{fair}_{\sigma_2(s_{\mathtt{init}})} \Rightarrow D^{\#}(\widetilde{w}', \widetilde{v}'))]\\ \to^*_{P'} \mathtt{fail}\end{array}$$

Here, $\widetilde{w}' = (w_1', \ldots, w_k')$ and $\widetilde{v}' = (v_1', \ldots, v_k')$ with $v_i' = \lceil v_i\rceil_V$. Note that $\mathtt{fair}_{\sigma_2(s_{\mathtt{init}})}$ evaluates to $\mathtt{true}$ by the condition $\sigma_2 \models_{\mathtt{fin}} C$, and $D^{\#}((w_1', \ldots, w_k'), (v_1', \ldots, v_k'))$ evaluates to $\mathtt{false}$ by the assumption $((w_1, \ldots, w_k), (v_1, \ldots, v_k)) \notin D$. □

4.3 Proof of Completeness (Theorem 4.3)

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 lefthand side of a reduction rule in Section 2 with $E = [\,]$.

Lemma 4.8 Suppose $\lceil P\rceil_{f,D} = P'$. If e_1 is an instruction and $\lceil e_1\rceil_{s,w} \to^+_{P'} e'$, then there exist e_2 and ℓ such that $e_1 \xrightarrow{\ell}_P e_2$ and either (i) $e' \to^+_{P'} \lceil e_2\rceil_{s',w'}$ is obtained without using the rule for function calls, or (ii) $\lceil e_1\rceil_{s,w} \to^+_{P'} \lceil e_2\rceil_{s',w'} \to^*_{P'} e'$. Furthermore, either (iii) $s' = \ell(s)$ and $w' = w$, or (iv) $s' = s_{\mathtt{init}}$ and $w' = (\lceil v_1\rceil_V, \ldots, \lceil v_k\rceil_V)$ with $e_1 = \langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k$.

Proof The proof proceeds by induction on the length of the reduction sequence $\lceil e_1\rceil_{s,w} \to^+_{P'} e'$, with a case analysis on the shape of e_1. We discuss only the main cases; the other cases are straightforward.

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

  $$\lceil e_1\rceil_{s,w} = (s\{A := \mathtt{true}\}, \star).$$

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

- Case $e_1 = \langle g\, v_1\ \cdots\ v_{i-1}\rangle v_i$: In this case, we have

  $$\lceil e_1\rceil_{s,w} = \langle g^{(i-1)}\lceil v_1\rceil_V \cdots \lceil v_{i-1}\rceil_V\rangle\ s\ w\, \lceil v_i\rceil_V.$$

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

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

  If $i = \mathtt{arity}(g)$ and $g = f$ with $g\, x_1\ \cdots\ x_i = e \in P$, the sequence $\lceil e_1\rceil_{s,w} \to^+_{P'} e'$ must be a prefix or postfix of the reduction sequence $\lceil e_1\rceil_{s,w} \to^+_{P'} \lceil[v_1/x_1, \ldots, v_n/x_n]e\rceil_{s,w}$ or $\lceil e_1\rceil_{s,w} \to^+_{P'} \lceil[v_1/x_1, \ldots, v_n/x_n]e\rceil_{s_{\mathtt{init}},(\lceil v_1\rceil_V, \ldots, \lceil v_k\rceil_V)}$. Thus, the result holds also for $e_2 = [v_1/x_1, \ldots, v_n/x_n]e$ and $\ell = \epsilon$.

□

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.

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

- $\lceil e_1 \rceil_{s,w} \rightarrow^*_{P'} E_1[\lceil \langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k \rceil_{s'',w''}]$ and
- $\lceil \langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k \rceil_{s'',w''}] \rightarrow^*_{P'} E_2[\lceil e_2 \rceil_{s',w'}]$

with $E = E_1[E_2]$ and $w' = (\lceil v_1 \rceil_V, \ldots, \lceil v_k \rceil_V)$.

Proof This follows by induction on the length of the reduction sequence $\lceil e_1 \rceil_{s,w} \rightarrow^*_{P'} E[\lceil e_2 \rceil_{s',w'}]$. We first note that $\lceil e_1 \rceil_{s,w} \rightarrow^+_{P'} E[\lceil e_2 \rceil_{s',w'}]$, since $w \neq w'$. Since $w \neq w'$, $\lceil e_1 \rceil_{s,w}$ must be reducible. So, e_1 can be decomposed to $E_3[e'_1]$ where e'_1 is an instruction. By the assumption $\lceil e_1 \rceil_{s,w} \rightarrow^*_{P'} E[\lceil e_2 \rceil_{s',w'}]$, we have one of the following cases.

(a) $\lceil e'_1 \rceil_{s,w} \rightarrow^+_{P'} (s_1, v)$ with $\lceil E_3 \rceil_w[v] \rightarrow^*_{P'} E[\lceil e_2 \rceil_{s',w'}]$.

(b) $\lceil e'_1 \rceil_{s,w} \rightarrow^+_{P'} E_4[\lceil e_2 \rceil_{s',w'}]$ with $E = \lceil E_3 \rceil_w[E_4]$.

In case (a), by repeated applications of Lemma 4.8, it follows that $v = \lceil v_1 \rceil_V$ for some v_1. Thus, we have $\lceil E_3[v] \rceil_{s_1,w} \rightarrow^*_{P'} E[\lceil e_2 \rceil_{s',w'}]$, and the required property follows from the induction hypothesis.

In case (b), by Lemma 4.8, there exist e'_2, s'', w'' such that $\lceil e'_1 \rceil_{s,w} \rightarrow^+_{P'} \lceil e'_2 \rceil_{s'',w''}$ and $\lceil e'_2 \rceil_{s'',w''} \rightarrow^*_{P'} E_4[\lceil e_2 \rceil_{s',w'}]$; notice that case (i) of Lemma 4.8 does not hold, since e_2 is an application. If the condition (iv) of Lemma 4.8 holds and $w'' = w'$, then we have the required condition. Otherwise, we can apply the induction hypothesis to get the required result. □

Proof of Theorem 4.3 Suppose $\lceil P \rceil_{f,D} = P'$ and $\mathtt{main}\, s_{\mathtt{init}} \perp \star \rightarrow^*_{P'} \mathtt{fail}$. Then, by Lemma 4.9, we have:

$$\begin{array}{l}
\mathtt{main}\, s_{\mathtt{init}} \perp \star \\
\rightarrow^*_{P'} E'_1[\langle f^{(k-1)}\ w'_1\ \cdots\ w'_{k-1}\rangle s\ w''\ w'_k] \\
\rightarrow^*_{P'} E'_1[[w'_1/x_1, \ldots, w'_k/x_k]\lceil e \rceil_{s_{\mathtt{init}},(w'_1,\ldots,w'_k)}] \\
\rightarrow^*_{P'} E'_1[E'_2[\langle f^{(k-1)}\ v'_1\ \cdots\ v'_{k-1}\rangle s'\ (w'_1, \ldots, w'_k)\ v'_k]] \\
\mathtt{fair}_{s'} \rightarrow^* \mathtt{true} \\
D^{\#}((w'_1, \ldots, w'_k), v'_1, \ldots, v'_k) \rightarrow^* \mathtt{false}
\end{array}$$

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

$$\mathtt{assert}(\mathtt{fair}_s \Rightarrow D^{\#}(w_k, (x_1, \ldots, x_k))).$$

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

$$\begin{array}{l}
\mathtt{main} \star \xrightarrow{\sigma_1}{}^*_P E_1[\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k] \\
\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k \xrightarrow{\sigma_2}{}^*_P E_2[\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k] \\
\lceil w_i \rceil_V = w'_i \qquad \lceil v_i \rceil_V = v'_i \qquad s' = \sigma_2(s_{\mathtt{init}})
\end{array}$$

By the conditions $\mathtt{fair}_{s'} \rightarrow^* \mathtt{true}$ and $s' = \sigma_2(s_{\mathtt{init}})$, we have $\sigma_2 \models_{\mathtt{fin}} C$. Together with the condition $\langle f\ w_1\ \cdots\ w_{k-1}\rangle w_k \xrightarrow{\sigma_2}{}^*_P E_2[\langle f\ v_1\ \cdots\ v_{k-1}\rangle v_k]$, we have $f\ w_1\ \cdots\ w_k \rhd^C_P f\ v_1\ \cdots\ v_k$. By the conditions $D^{\#}((w'_1, \ldots, w'_k), v'_1, \ldots, v'_k) \rightarrow^* \mathtt{false}$, $\lceil w_i \rceil_V = w'_i$, and $\lceil v_i \rceil_V = v'_i$, we have

$$((w_1, \ldots, w_k), (v_1, \ldots, v_k)) \notin D.$$

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 programs to OCaml, and we translate the property to fair termination (for ones from [20, 23] where the properties are given as temporal logic formulas or automaton inclusion problems). We have conducted the experiments on a machine with Intel Core i7-3930K (3.20 GHz, 16 GB of memory), with timeout of 600 seconds. The web interface of the implementation and the benchmarks used in the experiments are available on the web.[6]

program	cycle1	cycle2	time
intro	3	14	11.492
repeat	4	12	2.276
closure	6	18	9.76
hofmann-1 [13, 14]	2	4	0.232
hofmann-2 [13, 14]	3	8	1.032
koskinen-1 [20]	7	27	43.344
koskinen-2 [20]	5	16	3.412
koskinen-3-1 [20]	6	17	2.752
koskinen-3-2 [20]	4	14	2.216
koskinen-3-3 [20]	6	23	4.964
koskinen-4 [20]	10	35	132.552
lester [23]	8	36	38.356

Table 1. The experiment results.

Table 1 summarizes the experiment results. The column "program" shows the names of the programs. The columns "cycle1" and "cycle2" show the number of counterexample-guided refinement (CEGAR) iterations. MoCHi [19] itself uses CEGAR internally, and "cycle2" is the cumulative count of MoCHi's internal CEGAR iterations over the course of the verification process, whereas "cycle1" shows only the number of the outer CEGAR iterations (i.e., the number of times candidate termination arguments are inferred – cf. Figure 2). The column "time" shows the running time in seconds.

All of the benchmarks are fair terminating. We remark that *none of them can be verified by the previous automatic methods*. This is because the methods for imperative programs [10, 17, 22, 31] cannot handle functions, and the methods for higher-order functional programs cannot handle infinite data [18, 23–25] or can only handle safety properties [15, 19, 29, 34–36, 38, 39] or termination [10, 17, 22, 31]. As we describe below, the benchmarks from the previous literature [13, 20, 23] are either verified manually or verified after a manual finite-data abstraction in the respective works. Our work is the first fully automatic approach that is able to verify these instances.

We describe the benchmarks. The benchmark `intro` is P_{ex} from Section 1.1 with the fairness constraint $C' = \{(\mathsf{f}\ 0, \mathsf{false})\}$. The benchmark `repeat` is the one from Example 2.2. The benchmark `closure` is the program below with the fairness constraint $C = \{(A, \mathtt{Never})\}$.

```
const x y = x
finish x = event A; finish x
f g = let n = g ⋆ in
      if n > 0 then f (const (n − 1)) else finish ⋆
main x = let n = rand() in f (const n)
```

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/

cess even in the other benchmarks – such as when checking the reachability against the transformed program).

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 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

[1] A. M. Ben-Amram. General size-change termination and lexicographic descent. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2002.

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

[3] B. Cook and E. Koskinen. Making prophecies with decision predicates. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410. ACM, 2011.

[4] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 219–230. ACM, 2013.

[5] B. Cook, E. Koskinen, and M. Y. Vardi. Temporal property verification as a program analysis task. In Gopalakrishnan and Qadeer [11], pages 333–348.

[6] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 415–426. ACM, 2006.

[7] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.

[8] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[9] K. Fujima, S. Ito, and N. Kobayashi. Practical alternating parity tree automata model checking of higher-order recursion schemes. In C. Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2013.

[10] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Prog. Lang. Syst.*, 33(2):7:1–7:39, Feb. 2011.

[11] G. Gopalakrishnan and S. Qadeer, editors. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

[12] T. A. Henzinger and D. Miller, editors. *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. ACM, 2014.

[13] M. Hofmann and W. Chen. Abstract interpretation from Büchi automata. In Henzinger and Miller [12], pages 51:1–51:10.

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

[15] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In Gopalakrishnan and Qadeer [11], pages 470–485.

[16] T. Johnsson. Lambda lifting: Treansforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[17] N. D. Jones and N. Bohr. Call-by-value termination in the untyped lambda-calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[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, 11-14 August 2009, Los Angeles, CA, USA*, 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 M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233. ACM, 2011.

[20] E. Koskinen and T. Terauchi. Local temporal reasoning. In Henzinger and Miller [12], pages 59:1–59:10.

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

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

[23] M. M. Lester, R. P. Neatherway, C.-H. L. Ong, and S. J. Ramsay. Model checking liveness properties of higher-order functional programs. URL `https://mjolnir.comlab.ox.ac.uk/papers/thors.pdf`, 2011.

[24] R. P. Neatherway and C. L. Ong. TravMC2: higher-order model checking for alternating parity tree automata. In N. Rungta and O. Tkachuk, editors, *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, pages 129–132. ACM, 2014.

[25] C. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 81–90. IEEE Computer Society, 2006.

[26] A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2005.

[27] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

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

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

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

[31] D. Sereni. *Termination analysis of higher-order functional programs*. PhD thesis, Magdalen College, 2006.

[32] C. Skalka and S. F. Smith. History effects and verification. In W. Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128. Springer, 2004.

[33] 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.

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

[35] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 277–288. ACM, 2009.

[36] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 75–86. ACM, 2013.

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

[38] N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014.

[39] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for Haskell. In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014.

[40] K. Yokoyama. Ramsey's theorem and termination analysis. In *JAIST RCSV-Logic Workshop*, 2014. `http://www.jaist.ac.jp/rcsv/event/20141030-eventws/yokoyama.pdf`.