Automatic Termination Verification for Higher-Order Functional Programs[⋆]

Takuya Kuwahara[1], Tachio Terauchi[2], Hiroshi Unno[3], and Naoki Kobayashi[4]

[1] University of Tokyo, kuwahara@is.s.u-tokyo.ac.jp
[2] Nagoya University, terauchi@is.nagoya-u.ac.jp
[3] University of Tsukuba, uhiro@cs.tsukuba.ac.jp
[4] University of Tokyo, koba@is.s.u-tokyo.ac.jp

Abstract. We present an automated approach to verifying termination of higher-order functional programs. Our approach adopts the idea from the recent work on termination verification via transition invariants (a.k.a. binary reachability analysis), and is fully automated. Our approach is able to soundly handle the subtle aspects of higher-order programs, including partial applications, indirect calls, and ranking functions over function closure values. In contrast to the previous approaches to automated termination verification for functional programs, our approach is sound and complete, relative to the soundness and completeness of the underlying reachability analysis and ranking function inference. We have implemented a prototype of our approach for a subset of the OCaml language, and we have confirmed that it is able to automatically verify termination of some non-trivial higher-order programs.

1 Introduction

Recent years have witnessed a dramatic progress in automated verification of higher-order functional programs [27, 31, 24, 11, 16, 33, 37]. The line of work takes the ideas from the recent advances in the verification of non-functional programs, such as predicate abstraction, counterexample-guided abstraction refinement, and interpolation [1, 9, 21, 10], to the verification of higher-order functional programs by way of refinement (dependent) types [36] and higher-order model checking [23, 14].

However, except for the case when the base-type data is of a finite domain [15, 20], the above line of work (to the extent of software model checking techniques for higher-order programs) has been limited to the verification of safety (i.e., reachability) properties. In particular, it is unable to verify liveness properties, such as termination.

For automated termination verification of higher-order programs, popular methods have been based on size-change termination [12, 29, 28] or TRS (term rewriting systems) techniques [6]. (Besides them, Xi [35] has proposed termination analysis based on dependent types, but his technique is not fully automated in the sense that users have to provide dependent types of recursive functions as witness of termination.) The current methods based on those approaches are not completely satisfactory, especially in terms of precision. Roughly, these techniques first construct a finite graph (called a

[⋆] This work was supported by MEXT Kakenhi 23220001, 23700026, 25280023, and 25730035.

```
let rec app f x () =
     if x>0 then app f (x-1) () else f x () in
let id () = () in
let rec g x = if x=0 then id else app g x in
let t = * in g t ()
```

Fig. 1. A non-terminating higher-order program.

static call graph [12, 29, 28] or a termination graph [6]) that over-approximates certain dependencies on termination, and then use techniques for first-order programs [19, 7] to show that there is no cyclic dependency. In these two-phase approaches to termination, information is often lost in the first phase; see Section 5 for more details.

In the present paper, we follow an approach based on transition invariants [26, 4, 5], and extend it to deal with higher-order functional programs. The transition-invariant-based approach has emerged as a powerful technique for verifying termination of first-order imperative programs [26, 4, 5]. The technique iteratively reduces the termination verification problem to the problem of checking the *binary* reachability of program transition relations. It then delegates the binary reachability checking to a reachability checker by encoding the problem as a plain reachability problem via a program transformation. Advantages of this approach are that termination arguments can be flexibly adjusted for each program by choosing an appropriate binary reachability relation, 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. For example, consider the program:

```
let f x = if p(x) then () else loop_forever()
```

Then, a call of `f` terminates if and only if `p(x)` is true, and the latter condition can be checked during the reachability verification. In the higher-order case, termination verification would be even more complicated since the condition can be passed as a parameter of `f`. This shows the advantage of reducing termination verification to reachability verification, where all the relevant information (such as value-dependent control flow and size change) is put together and precisely analyzed by taking advantage of the recent advance of reachability verification tools for higher-order programs.

The extension of the transition invariant-based technique for higher-order programs is non-trivial. To see why, let us consider the OCaml program P_0 shown in Figure 1. (Here, $\star$ denotes a non-deterministic integer.) The program is non-terminating for any non-deterministic choice of t such that $\mathtt{t} < 0$. For example, for $\mathtt{t} = -1$, the program exhibits the following infinite reduction sequence.

$$\mathtt{g\ (-1)\ ()} \longrightarrow^* \mathtt{app\ g\ (-1)\ ()} \longrightarrow^* \mathtt{g\ (-1)\ ()} \longrightarrow^* \mathtt{app\ g\ (-1)\ ()} \longrightarrow^* \cdots$$

Note here that `g` is passed to and indirectly called by `app`, and there is no direct call to `g` in the definition of `g`. Moreover, `g` itself does not (totally) call `app` but returns a partially applied closure of the form `app g` n. Therefore, a termination verifier must soundly handle indirect calls and function closures to avoid incorrectly reporting that the program is terminating.

For a terminating example, let us consider a variation P_1 of the above program, obtained by replacing the branching condition $\mathtt{x} = 0$ in $\mathtt{g}$ with $\mathtt{x} \leq 0$. To prove that P_1 is terminating for any non-deterministically chosen t (and any non-deterministic choice of the integers chosen inside $\mathtt{g}$), we need to know that the sequence of the third arguments passed to the recursive calls to `app` is strictly decreasing and is bounded below by 0.

Our technique consists of: (i) a method to find an appropriate (disjunctively) well-founded relation, including those over function closure values, that over-approximates the binary reachability relation (the relation between two recursive function calls), (ii) a program transformation that reduces the binary reachability problem to the plain reachability problem, and (iii) a plain reachability analysis for higher-order programs. For (iii), we employ off-the-shelf reachability verification tools for higher-order programs [27, 31, 11, 16, 33, 37]. For (i), we adopt the previous technique [33] for automatically inserting implicit integer parameters that represent information about function closures. We can then adopt the existing techniques to find ranking functions (on integer arguments) from counterexamples [25, 3, 5]. The most subtle part is (ii): how to reduce the binary reachability analysis to plain reachability analysis. Actually, Ledesma-Garza and Rybalchenko [18] has recently tackled this problem, but (as admitted in [18], Section 8), their solution does not work quite well in the presence of partial applications and indirect function calls. In fact, their method cannot properly deal with the programs P_0 and P_1 above (cf. Section 5 of this paper for more details). By contrast, the reduction from binary reachability to plain reachability presented in this paper is *sound* and *complete*.

Our contributions are: (i) The first sound approach to the termination verification of higher-order functional programs that is based on the transition invariant / binary reachability technique. The approach is also complete relative to the completeness of the backend reachability checker and the ranking function inference process. A notable aspect of our approach is an inference of ranking functions over closure values via the automatic inference of implicit parameter instantiations. (ii) A prototype implementation to show the effectiveness of the proposed approach.

The rest of the paper is organized as follows. We define the target functional language of termination verification in Section 2. Section 3 formalizes our termination verification method. Section 4 reports on a preliminary implementation and experiment results. We compare our method with related work in Section 5 and conclude the paper in Section 6. Appendix contains extra materials and proofs of the theorems.

2 Preliminaries

In this section, we introduce a higher-order functional language L, which is the target of our termination verification. Figure 2 shows the syntax of L. Here, $\widetilde{x}$ is an abbreviation for a (non-empty) variable sequence $x_1\ x_2\ \ldots\ x_k$. The meta-variables f, x, c and op range over the sets of function symbols, variables, constants, and binary operators respectively. We write $|\widetilde{x}|$ for the length of $\widetilde{x}$. The arity of function f_i, written $arity(f_i)$, is the number of formal parameters, i.e., $|\widetilde{x_i}|$ in the function definition $f_i\ \widetilde{x_i} = e_i$. We assume that the set of constants includes $()$, $\mathtt{true}$, $\mathtt{false}$ and (unbounded) integers, and that the set of binary operators includes comparators: $>$, $<$, $\geq$, $\leq$, $=$ and boolean

$$
\begin{aligned}
\textit{Programs } P &::= \{f_1\ \widetilde{x}_1 = e_1, \ldots, f_n\ \widetilde{x}_n = e_n\} \\
\textit{Expressions } e &::= v \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e_1\ \mathit{op}\ e_2 \mid e_1\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid *_{\mathbf{int}} \\
\textit{Values } v &::= c \mid f \mid f\ \widetilde{v}\ (\text{where } |\widetilde{v}| < \mathit{arity}(f))
\end{aligned}
$$

Fig. 2. Syntax of L

$$
\begin{aligned}
\textit{Eval. contexts } E ::= {} & [\,] \mid E\ \mathit{op}\ e \mid v\ \mathit{op}\ E \mid E\ e \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \\
& \mid\ \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{aligned}
$$

$$E\,[c_1\ \mathit{op}\ c_2] \to_P E\,[[\![\mathit{op}]\!](c_1, c_2)] \qquad \frac{n \in \mathbb{Z}}{E\,[*_{\mathbf{int}}] \to_P E\,[n]}$$

$$E\,[\texttt{let}\ x = v\ \texttt{in}\ e] \to_P E\,[[v/x]\,e]$$

$$E\,[\texttt{if true then}\ e_1\ \texttt{else}\ e_2] \to_P E\,[e_1] \qquad \frac{f\ \widetilde{x} = e \in P \qquad |\widetilde{x}| = |\widetilde{v}|}{E\,[f\ \widetilde{v}] \to_P E\,[[\widetilde{v}/\widetilde{x}]\,e]}$$

$$E\,[\texttt{if false then}\ e_1\ \texttt{else}\ e_2] \to_P E\,[e_2]$$

Fig. 3. Operational semantics of L.

operators: $\wedge, \vee, \Rightarrow$. We also assume that a program has a special one-arity function named `main` that does not occur in the body of a function definition.

A program is a set of top-level function definitions; note that this does not lose generality because any functional program can be transformed to this form via λ-lifting. In the definition of the expression, $*_{\mathbf{int}}$ evaluates to some integer in a non-deterministic manner. Note that the non-deterministic boolean, $*_{\mathbf{bool}}$, can be defined as $*_{\mathbf{int}} = 0$.

The set of evaluation contexts and the reduction relation are given in Figure 3. Here, $[\![\mathit{op}]\!]$ denotes the binary operation on constants denoted by *op*. Note that the evaluation is call-by-value and non-deterministic (because of $*_{\mathbf{int}}$). We write $\to_P^*$ for the reflexive and transitive closure of $\to_P$, and $\to_P^+$ for the transitive closure of $\to_P$. When it is clear from the context, we omit the subscript P from the relations.

Example 1. The following program `fib` chooses an integer n non-deterministically and computes the n-th Fibonacci number.

$$\left\{\begin{array}{l} \quad \texttt{fib n} = \texttt{if n} < 2\ \texttt{then 1 else fib}(\texttt{n}-1) + \texttt{fib}(\texttt{n}-2), \\ \texttt{main}\ () = \texttt{fib}\ *_{\mathbf{int}} \end{array}\right\}$$

The following is a possible reduction sequence of the program:

$$
\begin{array}{l}
\texttt{main}\ () \to_{\texttt{fib}} \texttt{fib}\ *_{\mathbf{int}} \to_{\texttt{fib}} \texttt{fib}(2) \to_{\texttt{fib}}^* \texttt{fib}(1) + \texttt{fib}(0) \\
\to_{\texttt{fib}}^* 1 + \texttt{fib}(0) \to_{\texttt{fib}}^* 1 + 1 \to_{\texttt{fib}}^* 2
\end{array}
$$

For readability, we often write a program in the OCaml-like syntax as shown below:

```
let rec fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
let main () = fib *int
```

Example 2. The following program `indirect` is a simplified variant of the program P_1 in Section 1, obtained by removing the then branch of app and moving the decrement operation (i.e., $\mathtt{x} - 1$) to inside g.

$$\left\{\begin{array}{r}\mathtt{app\,f\,x\,u} = \mathtt{f\,x\,u} \\ \mathtt{id\,u} = \mathtt{u} \\ \mathtt{g\,x} = \mathtt{if\,x} \leq 0\,\mathtt{then\,id\,else\,app\,g}\,(\mathtt{x}-1) \\ \mathtt{main}\,() = \mathtt{g}\ *_{\mathbf{int}}\ ()\end{array}\right\}$$

The following is a possible reduction sequence of the program.

$$\begin{array}{l}\mathtt{main}\,() \to^* \mathtt{g}\,2\,() \to^* \mathtt{app\,g}\,1\,() \to^* \mathtt{g}\,1\,() \\ \to^* \mathtt{app\,g}\,0\,() \to^* \mathtt{g}\,0\,() \to^* \mathtt{id}\,() \to^* ()\end{array}$$

Note that, although g is applied to two arguments in the reduction above, $\mathit{arity}(\mathtt{g}) = 1$ in our definition (because x is the only formal parameter in the definition of g).

We define termination as follows.

Definition 1. *A program P is* terminating, *if there is no infinite reduction sequence* $\mathtt{main}\,() \to_P e_1 \to_P e_2 \to_P \cdots$.

Remark 1. The language L is untyped and therefore, a program evaluation may get stuck. 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 delegates the reachability checking to a higher-order program model checker for a typed language.

3 Termination Verification via Binary Reachability

This section describes our termination verification method. We give an informal overview of the whole process in Subsection 3.1, and discuss each step in a more detail in the later subsections.

3.1 Overview

We use `indirect` from Example 2 as a running example in this subsection. Our termination verification method is based on the observation that a functional program is terminating if and only if each of its *call tree*,[5] which expresses how the functions are called in an execution of the program, is finite. Figure 4 shows a call tree for `indirect`. Each node expresses a fully applied function call (a

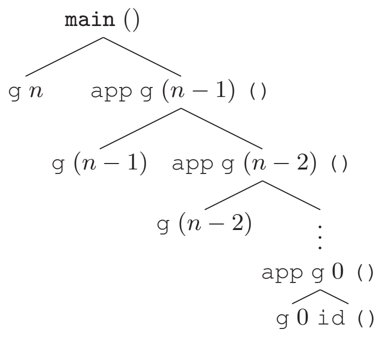

Fig. 4. A call tree of program `indirect`.

[5] The call tree in this paper roughly corresponds to the *dynamic call graph* of [29].

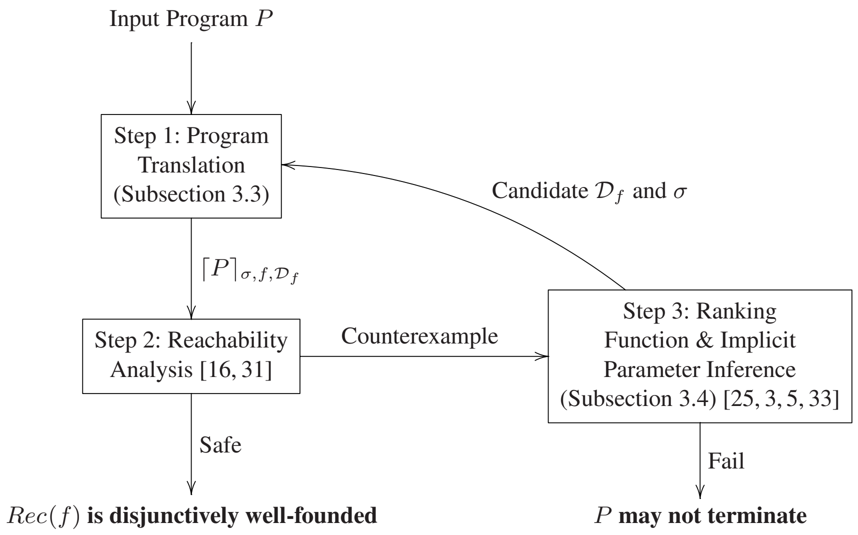
Input Program P
Step 1: Program Translation (Subsection 3.3)
$\lceil P \rceil_{\sigma,f,\mathcal{D}_f}$
Candidate $\mathcal{D}_f$ and σ
Step 2: Reachability Analysis [16, 31]
Counterexample
Step 3: Ranking Function & Implicit Parameter Inference (Subsection 3.4) [25, 3, 5, 33]
Safe
Fail
$Rec(f)$ is disjunctively well-founded
P may not terminate

Fig. 5. Overview of the termination verification process.

call of the form $f\,\widetilde{v}$ with $arity(f) = |\widetilde{v}|$),
and an edge represents that the function call represented by the target node is made from the function call of the source node. For example, the edge from "app g $(n-1)$ ()" to "g $(n-1)$" means that g $(n-1)$ is called during evaluation of app g $(n-1)$ () (i.e., app g $(n-1)$ () $\to^{+}$ $E[$g $(n-1)]$ for some evaluation context E). For every (possibly infinite) reduction sequence, the corresponding call tree is finitely branching. Therefore, by König's lemma, to show that every call tree of the program is finite and therefore the program is terminating, it is sufficient to show that no call tree has an infinite path from the root. The latter is equivalent to showing that for every function f, there is no infinite path $f\,\widetilde{v}_1 \triangleright f\,\widetilde{v}_2 \triangleright f\,\widetilde{v}_3 \triangleright \cdots$ where $f\,\widetilde{v} \triangleright g\,\widetilde{w}$ means that $f\,\widetilde{v}$ is an ancestor of $g\,\widetilde{w}$ in the call tree. In the running example, the only non-trivial sequence is app g n_1 () $\triangleright$ app g n_2 () $\triangleright$ app g n_3 () $\triangleright \cdots$ and this sequence must be finite since n_i is decreasing and bounded below by 0. Thus, we may conclude that `indirect` is terminating. We refer to Subsection 3.2 for the formal exposition of the above argument.

By the above argument, to show that the program terminates, it suffices to show that, for every function f, the relation $Rec(f) = \{(\widetilde{v}, \widetilde{v}') \mid f\,\widetilde{v} \triangleright f\,\widetilde{v}'\}$ is well-founded, or disjunctively well-founded[6] [26] because $Rec_P(f)$ is transitive. Thus, as mentioned in Section 1, our termination verification method proceeds as follows.

(i) For each function f, guess a disjunctively well-founded relation $\mathcal{D}_f$ that over-approximates $Rec(f)$, and reduce the termination verification problem to the so called the *binary reachability analysis* problem of showing $Rec(f) \subseteq \mathcal{D}_f$.

[6] A binary relation is *disjunctively well-founded* if it is a finite union of well-founded sets [26].

(ii) Use a program transformation to reduce the binary reachability problem to a plain reachability problem; and
(iii) Solve the plain reachability problem by using an off-the-shelf software model checker.

We express the disjunctively well-founded relation $\mathcal{D}_f$ by using a set of ranking functions, and gradually refine the set by using the technique of counterexample-guided abstraction refinement [25, 3, 5, 33].

Figure 5 shows the overall flow of the process. We start with an empty set of ranking functions (i.e., with $\mathcal{D}_f = \emptyset$), and apply a program transformation to reduce the binary reachability analysis problem of deciding $Rec(f) \subseteq \mathcal{D}_f$ to the plain reachability analysis problem of deciding if an assertion failure is reachable in the translated program $\lceil P \rceil_{\sigma,f,\mathcal{D}_f}$ (Step 1 in Figure 5). Here, σ is a *candidate implicit parameter instantiation* and is used for ranking functions over higher-order values (i.e., function closures). For the purpose of the exposition, we focus on the case where the ranking functions are only over the first-order values in Subsections 3.2 and 3.3, and defer implicit parameters and ranking functions over higher-order values to Subsection 3.4. For simplicity, we write $\lceil P \rceil_{f,\mathcal{D}_f}$ for $\lceil P \rceil_{\sigma,f,\mathcal{D}_f}$ when implicit parameters are irrelevant to the discussion.

Informally, the idea of the program transformation is to pass around the argument $\widetilde{pv}$ of an ancestor call as an *extra argument*, and assert in the body of f that $\widetilde{pv}$ and the current argument $\widetilde{v}$ are related by $\mathcal{D}_f$. For the running example, the definition of app would be transformed as follows.

```
app s1 f s2 x s3 u = assert((s3,(f,x,u))∈𝒟_app); ...
```

Here, s1, s2, and s3 are the extra arguments that carry the arguments of ancestor calls to app. Because a function body does not evaluate until the function is fully applied, only s3 is relevant and is checked with the current arguments $(\texttt{f}, \texttt{x}, \texttt{u})$ for the candidate disjunctively well-founded relation $\mathcal{D}_{\texttt{app}}$. We use non-determinism to ensure that the extra parameter can be bound to the argument of an *arbitrary* ancestor call, so that the transformed program is assertion safe if and only if $Rec(f) \subseteq \mathcal{D}_f$. Because a function can be called indirectly in a higher-order program, we pass extra arguments at *all* call sites, including indirect calls, to account for all possible ancestor relations. Therefore, for the running example, we also transform the functions g and id to take extra parameters (but only track and check the arguments of app). This makes our approach sound and complete even in the presence of higher-order functions.[7] In this manner, the binary reachability problem is reduced to a plain reachability problem. We refer to Subsection 3.3 for a more formal exposition.

We proceed to the plain reachability analysis (Step 2 in Figure 5), to check if the inserted assertion may fail. If the assertion cannot fail, then we conclude that $Rec(f) \subseteq \mathcal{D}_f$ holds (and if that is the case for every function f in the program, we conclude that the program is terminating). Even if the program is terminating, however, an assertion failure may occur, because of a wrongly guessed $\mathcal{D}_f$. For our running example, in the initial

[7] This is a crucial difference with the previous approaches to automated termination verification for higher-order programs [30, 28, 29, 6, 12] that handle higher-order functions by approximating the call graph up front (e.g., via a control flow analysis), which can lose precision when the calls depend on non-trivial safety conditions.

iteration, we set $\mathcal{D}_{\mathtt{app}} = \emptyset$, and when the transformed program is given to a reachability verifier (e.g., MoCHi [16]), the verifier would return a concrete counterexample to the assertion safety. Suppose the following counterexample is returned.

$$\mathtt{main}\ () \to^* \mathtt{app}\ s_1\ \mathtt{g}\ s_2\ 0\ (\mathtt{g}, 1, ())\ () \to^* \mathtt{fail}$$

which corresponds to the following reduction sequence in the original program:

$$\mathtt{main}\ () \to^* \mathtt{g}\ 2\ () \to^* \mathtt{app}\ \mathtt{g}\ 1\ () \to^* \mathtt{app}\ \mathtt{g}\ 0\ ()$$

We analyze the counterexample, and infer a new ranking function to refine $\mathcal{D}_f$ (Step 3 in Figure 5). For the running example, we look for a ranking function r such that $r(\mathtt{g}, 1, ()) > r(\mathtt{g}, 0, ()) \geq 0$ and update $\mathcal{D}_{\mathtt{app}}$ to $\mathcal{D}_{\mathtt{app}} \cup \{((pf, px, pu), (f, x, u)) \mid r(pf, px, pu) > r(f, x, u) \geq 0\}$. To find a ranking function, we adopt the existing techniques for inferring ranking functions from counterexamples of first-order programs [25, 5].[8] If no ranking function can be found, then we report that the program may not be terminating. This may happen either because the program is indeed non-terminating, or the method used for the ranking function synthesis is incomplete. We refer to Subsection 3.4 for a more formal exposition on this step.

Upon refining $\mathcal{D}_f$, we go back to Step 1 and repeat the process. If the program is terminating, and if the underlying reachability analysis tool and the ranking function synthesis *were* complete, the loop eventually terminates and we conclude that the program is terminating.

3.2 Termination and Binary Reachability

We discuss how termination verification is reduced to binary reachability analysis (to show $Rec(f) \subseteq \mathcal{D}_f$) more formally. First, we define the relations $\rhd$ and $Rec(f)$.

Definition 2. *The* call relation $\rhd_P$ *is the binary relation defined by:*

$$\begin{aligned}\rhd_P := \{(f\ \widetilde{v}, g\ \widetilde{w}) \mid (\mathtt{main}\ () \to_P^* E_1\,[f\ \widetilde{v}]) \wedge (f\ \widetilde{v} \to_P^+ E_2\,[g\ \widetilde{w}])\\ \wedge\, arity(f) = |\widetilde{v}| \wedge arity(g) = |\widetilde{w}|\}\end{aligned}$$

We often use the infix notation $f\ \widetilde{v} \rhd_P g\ \widetilde{w}$ for $(f\ \widetilde{v}, g\ \widetilde{w}) \in \rhd_P$.

Definition 3. $Rec_P(f)$*, the* recursion relation *of f in P, is the binary relation defined by:*

$$Rec_P(f) := \left\{(\langle\widetilde{v_1}\rangle, \langle\widetilde{v_2}\rangle) \,|\, f\ \widetilde{v_1} \rhd_P f\ \widetilde{v_2}\right\}$$

When it is clear from contexts, we omit the subscript P. Note that the relations $\rhd_P$ and $Rec_P(f)$ are transitive.

Example 3. Recall program `fib` in Example 1. $Rec(\mathtt{fib})$ is:

$$(\{(n, n-1) | n > 1\} \cup \{(n, n-2) \mid n > 2\})^+ = \{(m, n) \mid m > n \geq 0\}$$

[8] To infer a ranking relation over function closure values, we use implicit parameters and infer sufficient instantiations for them to represent the closures. This is done by adopting the counterexample-guided technique from the previous work [33] (cf. Subsection 3.4).

As shown in the example below, $Rec(f)$ may be non-empty even if f is not recursively defined, and $Rec(f)$ may be empty even if f is recursively defined.

Example 4. Recall the program in Example 2. The recursion relations are:

$$\begin{aligned} Rec(\mathtt{app}) &= \{(\langle \mathtt{g}, m, ()\rangle, \langle \mathtt{g}, n, ()\rangle) \mid m > n \geq 0\} \\ Rec(\mathtt{id}) &= Rec(\mathtt{g}) = \emptyset \end{aligned}$$

Note that g is defined recursively but does not cause a recursive call to g. Therefore, the relation $Rec(\mathtt{g})$ is empty. On the other hand, app is not recursively defined but $Rec(\mathtt{app}) \neq \emptyset$. This shows that we must check the disjunctive well-foundedness of $Rec(f)$ for each function f, regardless of whether f is recursively defined or not.

Next, we show the termination verification problem can be reduced soundly and completely to the problem of showing that $Rec(f)$ is disjunctively well-founded for every f.

We state the soundness and the completeness of the reduction.

Theorem 1 (Soundness). *A program P is terminating if $Rec_P(f)$ is disjunctively well-founded for every f defined in P.*

Theorem 2 (Completeness). *If a program P is terminating, then $Rec_P(f)$ is disjunctively well-founded for every f defined in P.*

3.3 From Binary Reachability to Plain Reachability

This subsection presents the reduction from the binary reachability problem of deciding $Rec_P(f) \subseteq \mathcal{D}_f$ to a plain reachability problem. As remarked in Subsection 3.1, we transform P to the program $\lceil P \rceil_{f,\mathcal{D}_f}$ that simulates the program P and asserts the property $(\widetilde{v'}, \widetilde{v}) \in \mathcal{D}_f$ whenever a recursive call relation $f\,\widetilde{v'} \rhd_P f\,\widetilde{v}$ is detected. Then, $Rec_P(f) \subseteq \mathcal{D}_f$ holds if and only if an assertion in $\lceil P \rceil_{f,\mathcal{D}_f}$ may fail.

The target language of the program transformation is an extension of L with tuples $\langle e_1, \ldots, e_k \rangle$, assertions $\mathtt{assert}(e_1); e_2$, and a special value $\bot$. The semantics of assertions is defined by:

$$E[\mathtt{assert}(\mathtt{true}); e] \to_P E[e] \qquad E[\mathtt{assert}(\mathtt{false}); e] \to_P \mathtt{fail}$$

where the evaluation contexts are extended accordingly: $E ::= \cdots \mid \mathtt{assert}(E); e$. The special value $\bot$ is used in place of the argument of an ancestor call, when there is no tracked ancestor call; see examples below.

Before we give the formal definition of the transformation, $\lceil \cdot \rceil_{f,\mathcal{D}_f}$, we informally describe the idea. The program $\lceil P \rceil_{f,\mathcal{D}_f}$ is obtained by adding extra function arguments that represent the arguments of past calls to f.

First we consider the simple case where the program only contains first-order functions. For example, let P be the Fibonacci program from Example 1. Let $\mathcal{D}_{\mathtt{fib}} = \{(\mathtt{pn}, \mathtt{n}) \mid \mathtt{pn} > \mathtt{n} \geq 0\}$. Then, $\lceil P \rceil_{\mathtt{fib},\mathcal{D}_{\mathtt{fib}}}$ would be as follows.

```
1: let rec fib pn n =
2:  assert(pn>n && n≥0);
3:  let pn' = if *bool then pn else n in
4:  if n < 2 then 1 else fib pn' (n-1) + fib pn' (n-2)
5: let main () = fib ⊥ *int
```

We have added the formal argument pn that represents the argument of an ancestor call (i.e., a call that corresponds to an ancestor node in the call tree) for fib, and inserted the assertion that checks $(\texttt{pn}, \texttt{n}) \in \mathcal{D}_{\texttt{fib}}$ (lines 1-2). Accordingly, we have also inserted an extra parameter to each function call (calls in line 4). In line 3, we non-deterministically "update" the tracked argument to the current argument in order to compare the argument of the current call with a future call of fib. The extra parameter is initially set to $\perp$, to indicate that there is no ancestor call (line 5). We assume that $>$ is extended so that $\perp > n$ holds for every n. For the two recursive calls to fib in the definition of fib, pn or n is passed in a non-deterministic manner. Below is a possible reduction of the transformed program.

$$
\begin{aligned}
&\texttt{main}\,() \to^* \texttt{fib}\,\perp 2 \to^* \texttt{assert}(\perp > 2 \,\&\&\, 2 \geq 0); \cdots \\
&\to^* \texttt{fib}\,2\,1 + \texttt{fib}\,2\,0 \\
&\to^* (\texttt{assert}(2 > 1 \,\&\&\, 1 \geq 0); \cdots) + \texttt{fib}\,2\,0 \\
&\to^* 1 + \texttt{fib}\,2\,0 \\
&\to^* 1 + (\texttt{assert}(2 > 0 \,\&\&\, 0 \geq 0); \cdots) \\
&\to^* 1 + 1 \to^* 2
\end{aligned}
$$

The subexpressions $\texttt{fib}\ 2\ 1$ and $\texttt{fib}\ 2\ 0$ capture the fact that $\texttt{fib}\ 1$ and $\texttt{fib}\ 0$ are called from $\texttt{fib}\ 2$ in the original program. It is easy to see that $Rec_P(\texttt{fib}) \subseteq \mathcal{D}_{\texttt{fib}}$ if and only if $\lceil P \rceil_{\texttt{fib},\mathcal{D}_{\texttt{fib}}}$ does not cause an assertion failure.

The transformation is more subtle for higher-order programs with partial applications and indirect calls. For example, let P be the program indirect from Example 2 and 4. Suppose that we wish to show $Rec_P(\texttt{app}) \subseteq \mathcal{D}_{\texttt{app}}$ where $\mathcal{D}_{\texttt{app}} = \{((\texttt{ph}, \texttt{pv}, \texttt{pu}), (\texttt{h}, \texttt{v}, \texttt{u})) \mid \texttt{pv} > \texttt{v} \geq 0\}$. Then, $\lceil P \rceil_{\texttt{app},\mathcal{D}_{\texttt{app}}}$ would be as follows.

```
 1: let app _ h _ v (ph,pv,pu) u =
 2:   assert(pv>v && v≥0);
 3:   let (ph,pv,pu) = if *bool then (ph,pv,pu) else (h,v,u)
 4:   in
 5:     h (ph,pv,pu) v (ph,pv,pu) u
 6: let id (ph,pv,pu) u = u
 7: let rec g (ph,pv,pu) x =
 8:   if x≤0 then id
 9:   else app (ph,pv,pu) g (ph,pv,pu) (x-1)
10: let main () = g ⊥̃ *int ⊥̃ ()
```

Here, $\widetilde{\perp}$ denotes $(\perp, \perp, \perp)$. As before, the extra parameter (ph,pv,pu) is inserted to represent the arguments of an ancestor call to app (line 1). Note that the extra parameter for this program takes a tuple of values, so that all three arguments of app can be tracked (i.e., h, v, and u). As before, the assertion is inserted at the beginning of the body of app to check $((\mathtt{ph}, \mathtt{pv}, \mathtt{pu}), (\mathtt{h}, \mathtt{v}, \mathtt{u})) \in \mathcal{D}_{\mathtt{fib}}$ (line 2), and we update the tracked past arguments with the current arguments non-deterministically, in order to compare the arguments of the current call with a future call of the function (line 3).

To soundly and completely track the extra parameters through partial applications and indirect calls, we pass the extra parameter at *every* function application site (lines 5, 9, and 10). To be able to do this, note that we have also transformed the definitions of id and g to take the extra parameters and pass them at the applications that occur in their body (lines 6 and 7), and also transformed the definition of app to take an extra parameter argument just before h and v as well as just before u, even though app only checks the well-foundedness against the one passed just before u. (Note that _ in the definition of app is an unused argument.) This is needed, because, in general, we cannot statically decide which (indirect) function call is a fully applied function call, nor which is a call to the target function (i.e., app in the example). We note that it is possible to soundly eliminate some of the redundancy via a static analysis (see Example 5 below), but it is in general impossible to completely decide *a priori* which function is called in what context. In effect, the idea of our transformation is to delegate such tasks to the backend reachability checker.

Example 5. By using useless code elimination [34, 13], we can simplify the above program to:

```
let app h v (ph,pv,pu) u =
  assert(pv>v && v≥0);
  let (ph,pv,pu) = if *bool then (ph,pv,pu) else (h,v,u) in
  h v (ph,pv,pu) u
let id (ph,pv,pu) u = u
let rec g x = if x≤0 then id else app g (x-1)
let main () = g *int ⊥̃ ()
```

Below is a possible reduction of $\lceil P \rceil_{\mathtt{app}, \mathcal{D}_{\mathtt{app}}}$. (For simplicity, we use a reduction sequence from the optimized version in Example 5.)

$$\begin{aligned}&\mathtt{main}\,() \longrightarrow^* \mathtt{g}\,2\,\widetilde{\bot}\,() \longrightarrow^* \mathtt{app}\,\mathtt{g}\,1\,\widetilde{\bot}\,() \\ &\longrightarrow^* \mathtt{g}\,1\,(\mathtt{g},1,())\,() \longrightarrow^* \mathtt{app}\,\mathtt{g}\,0\,(\mathtt{g},1,())\,()\end{aligned}$$

Note that the reached state $\mathtt{app}\,\mathtt{g}\,0\,(\mathtt{g},1,())\,()$ captures the recursion relation $\mathtt{app}\,\mathtt{g}\,1\,() \rhd_P \mathtt{app}\,\mathtt{g}\,0\,()$ of the original program P.

The Formal Definition of $\lceil\cdot\rceil_{f,\mathcal{D}}$

We now define the transformation formally. $\lceil P \rceil_{f,\mathcal{D}}$ is obtained by transforming each function definition:

$$\lceil P \rceil_{f,\mathcal{D}} = \{\lceil g\,\widetilde{x} = e \rceil_{f,\mathcal{D}} \mid g\,\widetilde{x} = e \in P\}$$

where the function definition transformation is defined as follows.

$$\lceil g\, x_1\, \cdots\, x_k = e \rceil_{f,\mathcal{D}} =$$
$$\begin{cases} g\, s_1\, x_1\, \cdots s_k\, x_k = \\ \quad \texttt{let}\ s = \mathbf{check}\&\mathbf{upd}(\mathcal{D}, s_k, \langle x_1, \dots, x_k\rangle)\ \texttt{in}\ \lceil e \rceil_s \\ \hfill \text{if } g = f \\ g\, s_1\, x_1\, \cdots s_k\, x_k = \lceil e \rceil_{s_k} \hfill \text{if } g \neq f \end{cases}$$

Note that an extra parameter s_i is added before every original parameter x_i of a function. Therefore, as opposed to the informal examples given above, the `main` function in the target program now takes *two* arguments, the first of which is always instantiated to $\perp$. The code that checks the candidate well-foundedness and non-deterministically updates the arguments is inserted at the beginning of the body of the target function f. Here, $\mathbf{check}\&\mathbf{upd}(\mathcal{D}, s_k, \langle x_1, \dots, x_k\rangle)$ denotes the expression

$$\texttt{assert}(\mathcal{D}^{\#}(s_k, \langle x_1, \dots, x_k\rangle)); \texttt{if}\ *_{\mathbf{bool}}\ \texttt{then}\ s_k\ \texttt{else}\ \langle x_1, \dots, x_k\rangle.$$

where the relation $\mathcal{D}^{\#}$ is the extension of $\mathcal{D}$ defined by:

$$\begin{aligned}\mathcal{D}^{\#} = &\{(\widetilde{\perp}, \langle v_1, \dots, v_k\rangle) \mid v_1, \dots, v_k \text{ are values}\} \cup \\ &\{(\langle v_1', \dots, v_k'\rangle, \langle v_1, \dots, v_k\rangle) \mid (\langle \lfloor v_1' \rfloor, \dots, \lfloor v_k' \rfloor\rangle, \langle \lfloor v_1 \rfloor, \dots, \lfloor v_k \rfloor\rangle) \in \mathcal{D}\}\end{aligned}$$

Here, $\lfloor v \rfloor$ is the value obtained by removing all the extra arguments from partial applications; see below for the definition. We assume that $\mathcal{D}^{\#}$ is represented by a formula of some logic. (In the implementation, we use the first-order logic with linear arithmetic.)

Note that the body e of each function definition is transformed by $\lceil e \rceil_s$ where s is the extra parameter passed just before the last argument (non-deterministically updated to the current arguments in the case of the target function). The expression transformation $\lceil e \rceil_s$ passes s at each application site in e, and is formally defined as follows.

$$\begin{aligned}&\lceil c \rceil_s = c \qquad \lceil *_{\mathbf{int}} \rceil_s = *_{\mathbf{int}} \qquad \lceil f \rceil_s = f \qquad \lceil x \rceil_s = x \\ &\lceil \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rceil_s = \texttt{let}\ x = \lceil e_1 \rceil_s\ \texttt{in}\ \lceil e_2 \rceil_s \\ &\lceil e_1\ op\ e_2 \rceil_s = \lceil e_1 \rceil_s\ op\ \lceil e_2 \rceil_s \\ &\lceil \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rceil_s = \texttt{if}\ \lceil e_1 \rceil_s \texttt{then}\ \lceil e_2 \rceil_s \texttt{else}\ \lceil e_3 \rceil_s \\ &\lceil e_1\ e_2 \rceil_s = \lceil e_1 \rceil_s\ s\ \lceil e_2 \rceil_s\end{aligned}$$

The operation $\lfloor e \rfloor$ for removing extra arguments (used in the definition of $\mathcal{D}^{\#}$) is defined as follows.

$$\begin{aligned}&\lfloor c \rfloor = c \qquad \lfloor *_{\mathbf{int}} \rfloor = *_{\mathbf{int}} \qquad \lfloor f \rfloor = f \qquad \lfloor x \rfloor = x \\ &\lfloor \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rfloor = \texttt{let}\ x = \lfloor e_1 \rfloor\ \texttt{in}\ \lfloor e_2 \rfloor \\ &\lfloor e_1\ op\ e_2 \rfloor = \lfloor e_1 \rfloor\ op\ \lfloor e_2 \rfloor \qquad \lfloor e_1\ s\ e_2 \rfloor = \lfloor e_1 \rfloor\ \lfloor e_2 \rfloor \\ &\lfloor \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rfloor = \texttt{if}\ \lfloor e_1 \rfloor \texttt{then}\ \lfloor e_2 \rfloor \texttt{else}\ \lfloor e_3 \rfloor\end{aligned}$$

Note that $\lfloor \lceil e \rceil_s \rfloor = e$.

We prove the soundness and the completeness of the transformation. The following theorem states the soundness of the transformation. It says that the target program reaches an assertion failure when the recursion relation is not a subset of $\mathcal{D}$.

Theorem 3 **(Soundness of $\lceil\cdot\rceil_{f,\mathcal{D}_f}$).**
Suppose that $\texttt{main}\,()\ \rightarrow_P^*\ E_1[f\,v_1\,\cdots\,v_k]$*,* $f\,v_1\,\cdots\,v_k\ \rightarrow_P^+\ E_2[f\,w_1\,\cdots\,w_k]$*, and* $(\langle v_1,\ldots,v_k\rangle,\langle w_1,\ldots,w_k\rangle)\notin\mathcal{D}$*. Then,* $\texttt{main}\,\bot\,()\rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}}\texttt{fail}$*.*

The theorem below states the completeness. It says that the target program reaches an assertion failure only when the recursion relation is not a subset of $\mathcal{D}$.

Theorem 4 **(Completeness of $\lceil\cdot\rceil_{f,\mathcal{D}_f}$).**
If $\texttt{main}\,\bot\,()\ \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}}\ \texttt{fail}$*, then* $\texttt{main}\,()\ \rightarrow_P^*\ E_1[f\,\widetilde{v}]$ *and* $f\,\widetilde{v}\ \rightarrow_P^+\ E_2[f\,\widetilde{w}]$*, and* $(\langle\widetilde{v}\rangle,\langle\widetilde{w}\rangle)\notin\mathcal{D}$ *for some* $E_1,E_2,\widetilde{v},\widetilde{w}$*.*

3.4 Ranking Function Inference

This subsection details how we refine the candidate disjunctively well-founded relation $\mathcal{D}_f$. As remarked in Subsection 3.1, we actually infer both $\mathcal{D}_f$ and the *implicit parameter instantiation* σ. The implicit parameters are used to assert and check well-founded relation over function closure values.

We first describe the case where only $\mathcal{D}_f$ is inferred. (This happens, for example, when f is first-order and does not take function closures as arguments.) Recall that the inference is invoked when $\mathit{Rec}(f)\not\subseteq\mathcal{D}_f$ (cf. Step 3 of Figure 5), and in such a case, the reachability checker returns a counterexample of the form:

$$\begin{aligned}\texttt{main}\,()\ \rightarrow^*\ & E[f\ s_1\ v_1\ s_2\ v_2\cdots(v_1',\ldots,v_n')\ v_n]\\ \rightarrow\ & \texttt{assert}((v_1',\ldots,v_n'),(v_1,\ldots,v_n))\in\mathcal{D}_f);\ldots\rightarrow\texttt{fail}\end{aligned}$$

As remarked in Subsection 3.3, this implies that $f\ v_1'\cdots v_n'\triangleright f\ v_1\cdots v_n$, and we have that $((v_1',\ldots,v_n'),(v_1,\ldots,v_n))\in\mathit{Rec}(f)$ and $((v_1',\ldots,v_n'),(v_1,\ldots,v_n))\notin\mathcal{D}_f$.

The goal of ranking function inference is to obtain a refined disjunctively well-founded relation $\mathcal{D}_f'$ such that

$$\mathcal{D}_f\cup\{((v_1',\ldots,v_n'),(v_1,\ldots,v_n))\}\subseteq\mathcal{D}_f'.$$

To this end, we infer a new ranking function $r(x_1,\ldots,x_n)$ such that $r(v_1',\ldots,v_n')>r(v_1,\ldots,v_n)\geq 0$ and let $\mathcal{D}_f'=\mathcal{D}_f\cup\{(\widetilde{x}',\widetilde{x})\mid r(\widetilde{x}')>r(\widetilde{x})\geq 0\}$. We adopt the constraint-based technique [25, 5] to infer $r(\widetilde{x})$.

We overview the inference process. We prepare a ranking function *template* $c_0+c_1x_1+\cdots+c_nx_n$. Here, c_i's are fresh variables, serving as *unknowns*. Then, we solve for the assignments to c_i's that satisfy the constraint

$$\forall\widetilde{x}.[\![\pi]\!]\Rightarrow c_0+c_1v_1'+\cdots+c_nv_n'>c_0+c_1v_1+\cdots+c_nv_n\geq 0$$

where $\widetilde{x}$ are the free variables in $[\![\pi]\!]$ and $v_1,\ldots,v_n,v_1',\ldots,v_n'$. Here, $[\![\pi]\!]$ is the strongest postcondition of the given counterexample π.[9] Finally, we set $r(x_1,\ldots,x_n)=\alpha_0+\alpha_1x_1+\cdots+\alpha_nx_n$ where each α_i is the assignment obtained for c_i.

Next, we extend the above process with *implicit parameters* to infer ranking functions over higher-order values. We illustrate the need for ranking functions over higher-order values with the following program `indirectHO`.

[9] More precisely, we construct a corresponding straightline program from the counterexample, and take its strongest postcondition (cf. [16, 33] and Appendix E).

```
let app h v = h () v
let id x = x
let rec g x u =
      if x <= 0 then id else app (g (x-1))
let main () = g *int () ()
```

The program is similar to `indirect` from Example 2 and 4, except that `app` no longer takes an integer argument and instead has the "decreasing" integer value captured inside the function closure passed as `h`. To show that this program is terminating, we need to show that the recursion relation for `app` is disjunctively well-founded. However, because `app` only takes function-type arguments (besides unit), ranking functions over first-order values are insufficient for this.

To this end, we adopt the idea from the previous work [33] and systematically add an integer-type *implicit parameter* just before each function-type parameter.[10] For `indirectHO`, we add an implicit parameter `h_IMPARAM` before `h` so that the program is now the following. (The added parts are underlined.)

```
let app h_IMPARAM h () = h () ()
let id x = x
let rec g x () =
      if x <= 0 then id else app $\underline{\sigma(\ell)}$ (g (x-1))
let main () = g *int () ()
```

Here, σ is the *candidate implicit parameter instantiation* that maps each *instantiation site* ℓ to an arithmetic expression over the variables bound in the context of ℓ. Formally, an instantiation site is at an application of a function-type argument, and is syntactically determined (i.e., between $e_1\ e_2$ where e_2 is function-type). Clearly, the addition of implicit parameters and their instantiations do not affect the termination of the program, and so we may check the termination of the program with the implicit parameters added to check the termination of the original. The verification process starts by initializing the candidate instantiations to some arithmetic expression (e.g., 0), and refine them iteratively via a counterexample analysis (cf. Figure 5).

As remarked above, in the presence of implicit parameters, we infer both σ and $\mathcal{D}_f$ when given a counterexample. To this end, the above inference process is extended as follows. We prepare templates for the instantiation expressions in addition to the templates for the ranking functions. Then, when generating the constraints, we use the template instantiation expressions in the strongest postcondition of the counterexample, and solve for both the unknowns in the ranking function templates and the instantiation expression templates.

More formally, let $\Pi = \{\pi_1, \ldots, \pi_m\}$ be the set of counterexamples we have seen so far for f. We prepare a template instantiation map Δ that maps each ℓ to an expression of the form $c_0 + c_1x_1 + \cdots + c_nx_n$ where c_i's are fresh unknowns and x_i's are the integer-type variables that are bound in the context of ℓ (which may include implicit

[10] Implicit parameters are called "extra parameters" in [33]. We call them implicit parameters here to avoid confusion with the extra parameters in this paper which are used for a different purpose.

program	ord	time	program	ord	time	program	ord	time
`Ackermann`	1	5.85	`alias_partial`	1	0.32	`churchNum`	4	3.13
`Fibonacci`	1	0.15	`quicksort`	2	timeout	`CE-Jones_Bohr`	4	0.71
`McCarthy91`	1	4.95	`indirectIntro`	2	4.76	`up_down`	2	0.65
`loop2`	1	0.61	`indirect`	2	1.36	`map`	2	1.59
`append`	1	0.14	`indirectHO`	2	7.75	`toChurch`	2	0.69
`zip`	1	0.15	`CE-0CFA`	2	0.14	`x_plus_2^n`	2	2.02
`binomial`	1	0.70	`CE-1CFA`	2	0.24	`foldr`	2	1.19

Table 1. Experiment results.

parameters). Then, we form the following constraint

$$\bigwedge_{\pi \in \Pi} \forall \widetilde{x}.[\![\pi \Delta]\!] \Rightarrow c_{\pi,0}+c_{\pi,1}v'_{\pi,1}+\cdots+c_{\pi,n}v'_{\pi,n} > c_{\pi,0}+c_{\pi,1}v_{\pi,1}+\cdots+c_{\pi,n}v_{\pi,n} \geq 0$$

where $c_{\pi,i}$'s are fresh unknowns, and $\widetilde{x}$ are the free non-unknown variables in $[\![\pi \Delta]\!]$ and $v_1, \ldots, v_n, v'_1, \ldots, v'_n$. (Here, we assume that the counterexamples π explicitly use the instantiation sites ℓ as expressions.) We solve for the unknowns that satisfy the constraint to obtain implicit parameter instantiations and ranking functions that refute the counterexample. We obtain the new candidate disjunctive well-founded relation from the ranking functions: $\mathcal{D} = \bigcup_{\pi \in \Pi}\{(\widetilde{x}', \widetilde{x}) \mid r_\pi(\widetilde{x}') > r_\pi(\widetilde{x}) \geq 0\}$ where $r_\pi(\widetilde{x}) = \alpha_{\pi,0} + \alpha_{\pi,1}x_1 + \cdots + \alpha_{\pi,n}x_n$ and each $\alpha_{\pi,i}$ is the obtained assignment for $c_{\pi,i}$. And, we substitute the assignments to the unknowns in Δ to obtain the new candidate implicit parameter instantiation. Appendix E contains details of the inference process applied to `indirectHO`.

4 Implementation and Experiments

We have implemented a prototype of the termination verifier for a subset of OCaml. We use MoCHi [16] as the backend reachability checker, and Z3 [22] as a constraint solver for ranking function inference. As an optimization, we have extended the ranking function inference described in Section 3.4 to also infer lexicographic linear ranking functions [5] whenever possible. Appendix F contains details of the lexicographic linear ranking function inference process.

We have tested our tool on various termination verification benchmark programs in literature, taken mostly from the previous work on termination verification of higher-order programs, as well as some synthetic but non-trivial examples. We ran the experiment on a machine with 3.20GHz CPU and 16GB of memory, with timeout of 600 seconds. The web interface of the verification tool and the programs used in the experiments are available online [17].

Table 1 summarizes the experiment results. The column "program" shows the names of programs, and the column "ord" shows the order of the program (where order-1 functions take only base type values, order-2 functions may take order-1 functions as arguments, etc., and the order of a program is the maximum order of the functions in the program). The column "time" shows the running time in seconds.

We briefly describe the benchmark programs. The seven programs in the left column and `alias_partial` are first-order (i.e., order-1) programs. The programs `append`, `zip`, and `binomial` are from [2]. `Ackermann` is the Ackermann's function, and is also used as examples in [2, 35]. `McCarthy91` is the McCarthy's 91 function (used as a benchmark program in, e.g., [18][11]). `Fibonacci` is the Fibonacci number function from Example 1. The program `alias_partial` is from Section 8 of [18] and is given as an example on which their approach fails.

The program `quicksort` is from [35],[12] and is a second-order program where the list sorting function is parametrized by the "compare" function. We check the termination of a program that passes the sorting function a terminating compare function and an arbitrary list. (Our tool currently does not directly support lists, and so a list is represented by the integer denoting its length.) Our tool fails to verify the program within the time limit due to the underlying reachability checker MoCHi failing to verify the necessary assertion safety. This is not a fundamental limitation with our termination verification approach, and we expect further advances in reachability verification to allow our approach to verify instances like `quicksort`.

The rest of the programs are higher-order programs whose termination depends non-trivially on the functions passed as the arguments, and precise reasoning about the function arguments is required for proving termination. They are mostly from the examples and benchmarks in [12, 28, 29]. Many of these are difficult examples that the previous approaches cannot verify. (We have selected the ones given as examples where their approaches fail). We refer to Appendix G for further description of these programs. As seen in Table 1, the benchmark results are promising and show that our tool is able to automatically verify the difficult instances quickly, except `quicksort` whose reason for the failure is elaborated above.

5 Related Work

There have been three major approaches to automated termination verification for first-order programs: *transition invariants* [26, 3, 4], *size-change termination* [19], and *term rewriting* [7] (see also [32, 8] for relationships between those approaches). The approaches have recently been extended to the termination verification of higher-order programs [18, 30, 28, 29, 6, 12]. Below, we compare them with our approach.

5.1 Transition Invariants

Closest to our work is the work by Ledesma-Garza and Rybalchenko [18]. Similar to our work, they propose a program transformation to reduce the transition invariant verification problem (i.e., binary reachability analysis) to a plain reachability problem via a program transformation. Unfortunately, as also admitted in their paper (Section 8 of [18]), their approach has a limited applicability to the verification of higher-order

[11] [18] is not fully automated and requires the user to provide the sufficient ranking functions as well as the predicates to be used for reachability checking.

[12] [35] is not automated.

programs because it does not correctly handle indirect calls and is actually unsound. For example, their approach would incorrectly report the program P_0 from Section 1 to be terminating. Moreover, their approach is not fully automated and requires a sufficient well-foundedness relation to be provided manually, and it also cannot handle well-foundedness relations over function closure values.

By contrast, we have proposed the first sound and (relatively) complete approach to termination verification of higher-order programs via binary reachability analysis. A key idea of our approach is the novel program transformation that precisely tracks the call-tree ancestor's arguments values through the higher-order control flow without a priori approximation. We have also presented a method to infer well-foundedness relations (including those over function closure values) from counterexamples returned by a higher-order program verifier, thus realizing a fully automated verification.

5.2 Size-Change Analysis

The size-change approach [19] to termination verification involves the following two steps: (1) an analysis of the program to construct a *size-change graph*, and (2) an analysis of the obtained graph to decide if the program is terminating. For functional programs, the size-change graph is a graph comprising functions in the program where the edges express the changes in the values that may be passed as arguments. Step (1) constructs the graph by statically approximating the possible calls that the program would make in its actual execution.

To apply size-change termination verification to higher-order programs, a control flow analysis (CFA) is employed to statically approximate the possible call relations as a call graph and construct a sound size-change graph from the call graph [30, 28, 29, 12]. Therefore, the approach involves *a priori* approximation of the program, and can lead to loss in a precision when a precise graph cannot be constructed by the static analysis. For example, the approach may fail on cases where a non-terminating call depends on a safety property (recall the simple example from Section 1 where a non-terminating function is called if and only if the condition $p(x)$ is met). By contrast, our approach suffers no a priori loss in precision and is sound and complete.

Like our approach, the size-change approach to higher-order programs [30, 28, 29, 12] can prove termination of programs that require well-foundedness relation over function closure values. For example, Jones and Bohr [12] and Sereni [28] order closure arguments by using the subtree relation on their tree representations. By contrast, we have presented a generic approach that uses implicit parameters and counterexample analysis to infer the appropriate instantiations for the implicit parameters. Our approach is more general in the sense that it is not fixed to one pattern of closure information to be used for the well-foundedness relation. For example, the subtree relation used in [12, 28] can be expressed by inferring instantiations that encode the depth of the closures, and our prototype implementation automatically verifies examples in their paper that require such information (cf. Section 4 and Appendix G).

On the other hand, we employ counterexample analysis and constraint-based inference to automatically infer the instantiations, and so the approach of Jones, Bohr, and Sereni that fixes the closure information to a pre-determined pattern may be more efficient on instances that are known to be verifiable with such information.

5.3 Term Rewriting

Similar to the size-change approach, the application of termination verification techniques for term rewriting systems to higher-order programs is done in a two-step process [6]. There, in the first step, a static analysis is employed to construct a term rewriting system that soundly approximates the given program such that the program is terminating if the constructed rewriting system is terminating. Then, the second step applies a termination verifier for term rewriting systems [7] to verify termination.

As with the size-change approach, this two-step approach can introduce a loss in precision because of the approximation in the first step. For example, Giesl et al. [6] show a simple program on which their approach fails because of this limitation (Example 4.12 in [6]).

6 Conclusion

We have presented a new automated approach to termination verification of higher-order functional programs. In stark contrast to the previous approaches, our approach is sound and complete relative to the soundness and completeness of the underlying reachability analysis and ranking function inference. Our approach is the first sound binary reachability analysis based approach to the termination verification of higher-order programs. The key features of our approach are the novel program transformation that correctly tracks the call-tree ancestor's arguments through the higher-order control flow, and the inference method for ranking functions over higher-order values via implicit parameter instantiation inference.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL. pp. 1–3 (2002)
2. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher-Order and Symbolic Computation 14(2-3), 261–300 (2001)
3. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 87–101. Springer (2005)
4. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI. pp. 415–426. ACM (2006)
5. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 47–61. Springer (2013)
6. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems 33(2), 7:1–7:39 (Feb 2011)
7. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: RTA. LNCS, vol. 3091, pp. 210–220. Springer (2004)
8. Heizmann, M., Jones, N.D., Podelski, A.: Size-change termination and transition invariants. In: SAS. LNCS, vol. 6337, pp. 22–50. Springer (2010)
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70 (2002)
10. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)

11. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying functional programs using abstract interpreters. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 470–485. Springer (2011)
12. Jones, N.D., Bohr, N.: Call-by-value termination in the untyped lambda-calculus. Logical Methods in Computer Science 4(1) (2008)
13. Kobayashi, N.: Type-based useless-variable elimination. Higher-Order and Symbolic Computation 14(2-3), 221–260 (2001)
14. Kobayashi, N.: Model checking higher-order programs. Journal of the ACM 60(3) (2013)
15. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS. pp. 179–188. IEEE Computer Society (2009)
16. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI. pp. 222–233. ACM (2011)
17. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs (2013), `http://www-kb.is.s.u-tokyo.ac.jp/~kuwahara/termination`
18. Ledesma-Garza, R., Rybalchenko, A.: Binary reachability analysis of higher order functional programs. In: SAS. LNCS, vol. 7460, pp. 388–404. Springer (2012)
19. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL. pp. 81–92. ACM (2001)
20. Lester, M.M., Neatherway, R.P., Ong, C.H.L., Ramsay, S.J.: Model checking liveness properties of higher-order functional programs. In: Proceedings of ML Workshop 2011 (2011)
21. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006)
22. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
23. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS. pp. 81–90. IEEE Computer Society (2006)
24. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL 2011. pp. 587–598. ACM (2011)
25. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. Lecture Notes in Computer Science, vol. 2937, pp. 239–251. Springer (2004)
26. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS. pp. 32–41. IEEE Computer Society (2004)
27. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI. pp. 159–169. ACM (2008)
28. Sereni, D.: Termination analysis of higher-order functional programs. Ph.D. thesis, Magdalen College (2006)
29. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In: ICFP. pp. 71–84. ACM (2007)
30. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: APLAS. Lecture Notes in Computer Science, vol. 3780, pp. 281–297. Springer (2005)
31. Terauchi, T.: Dependent types from counterexamples. In: POPL. pp. 119–130. ACM (2010)
32. Thiemann, R., Giesl, J.: The size-change principle and dependency pairs for termination of term rewriting. Appl. Algebra Eng. Commun. Comput. 16(4), 229–270 (2005)
33. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: POPL. pp. 75–86. ACM (2013)
34. Wand, M., Siveroni, I.: Constraint systems for useless variable elimination. In: Proceedings of POPL '99. pp. 291–302 (1999)
35. Xi, H.: Dependent types for program termination verification. In: LICS '01. pp. 231–242. IEEE (2001)

36. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL. pp. 214–227 (1999)
37. Zhu, H., Jagannathan, S.: Compositional and lightweight dependent type inference for ML. In: VMCAI. LNCS, vol. 7737, pp. 295–314. Springer (2013)

Appendix

A Proof of Theorem 1

We first prove the following lemma.

Lemma 1. *If $Rec_P(f)$ is disjunctively well-founded for every function f defined in P, then every sequence $f_1\ \widetilde{v_1} \rhd_P f_2\ \widetilde{v_2} \rhd_P \ldots \rhd_P f_n\ \widetilde{v_n} \rhd_P \ldots$ is finite.*

Proof. Suppose that $Rec_P(f)$ is disjunctively well-founded for every f, but that there is an infinite sequence $f_1\ \widetilde{v_1} \rhd_P f_2\ \widetilde{v_2} \rhd_P \ldots \rhd_P f_n\ \widetilde{v_n} \rhd_P \ldots$. Because the number of function symbols f_i is finite, there must be f such that $f_i = f$ for infinitely many i's. But, this contradicts with the assumption that $Rec_P(f)$ is disjunctively well-founded and the fact that $Rec_P(f)$ is transitive. (Recall that a transitive relation that is disjunctively well-founded is also well-founded.) □

We are now ready to prove the soundness of the reduction.

Proof. **[Theorem 1]** Suppose that $Rec_P(f)$ is disjunctively well-founded for every f, but that there were an infinite reduction sequence π:

$$\mathtt{main}\ ()(= e_0) \rightarrow_P e_1 \rightarrow_P e_2 \rightarrow_P \cdots$$

We write $(f\ \widetilde{v}, i) \rhd^{\pi}_P (g\ \widetilde{w}, j)$ if, for some E_1 and E_2, $e_i = E_1[f\ \widetilde{v}]$ and $e_j = E_1[E_2[g\ \widetilde{w}]]$ with $j > i \geq 0$, $arity(f) = |\widetilde{v}|$ and $arity(g) = |\widetilde{w}|$ and if E_1 is not reduced in the sub-sequence $e_i \rightarrow^*_P e_j$ of π. By definition, $\rhd^{\pi}_P$ is a restriction of $\rhd_P$ in the sense that $(f\ \widetilde{v}, i) \rhd^{\pi}_P (g\ \widetilde{w}, j)$ implies $f\ \widetilde{v} \rhd_P g\ \widetilde{w}$.

Define T_π as the tree consisting of the set V_π of nodes and the set E_π of edges given by:

$$\begin{aligned} V_\pi &= \{(f\ \widetilde{v}, i) \mid arity(f) = |\widetilde{v}| \text{ and } e_i = E\,[f\ \widetilde{v}]\} \\ E_\pi &= \{((f\ \widetilde{v}, i), (g\ \widetilde{w}, j)) \mid (f\ \widetilde{v}, i) \rhd^{\pi}_P (g\ \widetilde{w}, j) \\ &\qquad \wedge \neg\exists k, h, \widetilde{z}.((f\ \widetilde{v}, i) \rhd^{\pi}_P (h\ \widetilde{z}, k) \wedge k < j)\} \end{aligned}$$

We note that this is the formal definition of the call tree mentioned in Subsection 3.1.

By the assumption that π is infinite, function definitions must be unfolded infinitely many times; thus, the tree T_π must be infinite. However, T_π is finitely branching (to observe this, notice that if $((f\ \widetilde{v}, i), (g\ \widetilde{w}, j)) \in E_\pi$, then $(g\ \widetilde{w}, j)$ must be called from the body of f obtained by reducing $f\ \widetilde{v}$), and every path from the root in T_π must be finite, by Lemma 1 and the fact that $\rhd^{\pi}_P$ is a restriction of $\rhd_P$. By König's lemma, T_π must be finite, hence a contradiction. □

B Proof of Theorem 2

It suffices to show that $Rec_P(f)$ is well-founded if P is terminating. We prove the contraposition. Suppose that $Rec_P(f)$ is not well-founded. Then there is an infinite sequence $f\ \widetilde{v}_0 \rhd_P f\ \widetilde{v}_1 \rhd_P f\ \widetilde{v}_2 \rhd_P \ldots$. By the definition of $\rhd_P$, we have $f\ \widetilde{v}_i \rightarrow^+_P E_i\,[f\ \widetilde{v}_{i+1}]$ for every $i \geq 0$, and $\mathtt{main}\ () \rightarrow^*_P E_0\,[f\ \widetilde{v}_1]$. Therefore, we have an infinite reduction sequence:

$$\mathtt{main}\ () \rightarrow^*_P E_0\,[f\ \widetilde{v}_1] \rightarrow^+_P E_0\,[E_1\,[f\ \widetilde{v}_2]] \rightarrow^+_P E_0\,[E_1\,[E_2\,[f\ \widetilde{v}_3]]] \rightarrow^+_P \cdots$$

Thus, P is not terminating.

C Proof of Theorem 3

We define the relation $\sim_s$ between the source expressions and the target expressions as follows,

$$\frac{v \text{ is a value}}{\lfloor v \rfloor \sim_s v}$$

$$\frac{}{x \sim_s x}$$

$$\frac{}{*_{\mathbf{int}} \sim_s *_{\mathbf{int}}}$$

$$\frac{e_1 \sim_s e_1' \qquad e_2 \sim_s e_2'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \sim_s \texttt{let}\ x = e_1'\ \texttt{in}\ e_2'}$$

$$\frac{e_1 \sim_s e_1' \qquad e_2 \sim_s e_2'}{e_1\ op\ e_2 \sim_s e_1'\ op\ e_2'}$$

$$\frac{e_1 \sim_s e_1' \qquad e_2 \sim_s e_2' \qquad e_3 \sim_s e_3'}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \sim_s \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3'}$$

$$\frac{e_1 \sim_s e_1' \qquad e_2 \sim_s e_2'}{e_1\ e_2 \sim_s e_1'\ s\ e_2'}$$

Let $\mathcal{D}_0$ be the relation that relates all the elements, that is, $\mathcal{D}_0(\langle \widetilde{v} \rangle, \langle \widetilde{v'} \rangle)$ for all values $\widetilde{v}$ and $\widetilde{v}'$.

Lemma 2. *Let P be a program, e_1 be a closed expression, and s be $\perp$ or a tuple of values $\langle v \rangle$. If $e_1 \to_P e_2$ and $e_1 \sim_s e_1'$, then $e_1' \to^*_{\lceil P \rceil_{f,\mathcal{D}_0}} e_2'$ for some e_2' such that $e_2 \sim_s e_2'$.*

Proof. This follows by case analysis on the rule used for $e \to_P e'$. The only non-trivial case is when

$$e_1 \equiv E[f\,\widetilde{v}] \to_P E[[\widetilde{v}/\widetilde{x}]e_0] \equiv e_2$$

where $f\,\widetilde{x} = e_0$. By the assumption $e_1 \sim_s e_1'$, e_1' is of the form $E'[f\,s_1\,v_1'\,\cdots\,s_k\,v_k']$ with $s_k = s$, $E \sim_s E'$ and $\widetilde{v} \sim_s v_1' \cdots v_k'$ (where $\sim_s$ is extended to contexts in a natural way). Thus, we have:

$$\begin{aligned}
& e_1 \equiv E'[f\,s_1\,v_1'\,\cdots\,s_k\,v_k'] \\
& \to^* E'[\texttt{let}\ s_0 = \mathbf{check}\&\mathbf{upd}(\mathcal{D}_0^{\#}, s, \langle \widetilde{v'} \rangle)\ \texttt{in} \\
& \qquad\quad [v_1'/x_1, \ldots, v_k'/x_k]\lceil e_0 \rceil_{s_0}] \\
& \to^* E'[\texttt{let}\ s_0 = s\ \texttt{in}\ [v_1'/x_1, \ldots, v_k'/x_k]\lceil e_0 \rceil_{s_0}] \\
& \to^* E'[[v_1'/x_1, \ldots, v_k'/x_k]\lceil e_0 \rceil_s]
\end{aligned}$$

and $e_2 \sim_s E'[[v_1'/x_1, \ldots, v_k'/x_k]\lceil e_0 \rceil_s]$ as required. □

The following lemma states that every element of the recursion relation may occur in a reduction sequence of the target program.

Lemma 3. *Suppose* $\texttt{main}\,() \rightarrow_P^* E_1[f\,v_1\,\cdots\,v_k]$ *and* $f\,v_1\,\cdots\,v_k \rightarrow_P^+ E_2[f\,w_1\,\cdots\,w_k]$. *Then,*

$$\texttt{main}\,\bot\,() \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}_0}} E[f\,s_1\,w_1'\,\cdots\,s_k\,w_k']$$

for some E,$s_1,\ldots,s_k$ *and* $w_1',\ldots,w_k'$ *such that* $s_k = \langle v_1',\ldots,v_k'\rangle$ *with* $\lfloor v_i'\rfloor = v_i$ *and* $\lfloor w_i'\rfloor = w_i$ *for* $i \in \{1,\ldots,k\}$.

Proof. By the assumption $\texttt{main}\,() \rightarrow_P^* E_1[f\,v_1\,\cdots\,v_k]$ and Lemma 2, we have

$$\texttt{main}\,\bot\,() \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}_0}} e_1$$

for some e_1 such that $E_1[f\,v_1\,\cdots\,v_k] \sim_\bot e_1$. Thus, e_1 must be of the form $E_1'[f\,s_1\,v_1'\cdots\bot\,v_k']$ where $\lfloor v_i'\rfloor = v_i$ for every $i \in \{1,\ldots,k\}$. By the assumption $f\,v_1\,\cdots\,v_k \rightarrow_P^+ E_2[f\,w_1\,\cdots\,w_k]$, we have

$$f\,v_1\,\cdots\,v_k \rightarrow_P [v_1/x_1,\ldots,v_k/x_k]e_0 \rightarrow_P^* E_2[f\,w_1\,\cdots\,w_k]$$

where $f\,x_1\,\cdots\,x_k = e_0 \in P$. The expression $f\,s_1\,v_1'\cdots\bot\,v_k'$ is reduced as follows:

$$\begin{aligned}
&f\,s_1\,v_1'\cdots\bot\,v_k'\\
&\rightarrow_{\lceil P\rceil_{f,\mathcal{D}_0}} \texttt{let}\ s = \mathbf{check\&upd}(\mathcal{D}_0^{\#},\bot,\langle v_1',\ldots,v_k'\rangle)\ \texttt{in}\\
&\qquad [v_1'/x_1,\ldots,v_k'/x_k]\lceil e_0\rceil_s\\
&\rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}_0}} [v_1'/x_1,\ldots,v_k'/x_k]\lceil e_0\rceil_{\langle v_1',\ldots,v_k'\rangle}
\end{aligned}$$

Let $\widetilde{v'} = \langle v_1',\ldots,v_k'\rangle$. By

$$[v_1/x_1,\ldots,v_k/x_k]e_0 \rightarrow_P^* E_2[f\,w_1\,\cdots\,w_k]$$

and

$$[v_1/x_1,\ldots,v_k/x_k]e_0 \sim_{\widetilde{v'}} [v_1'/x_1,\ldots,v_k'/x_k]\lceil e_0\rceil_{\widetilde{v'}},$$

we have:

$$[v_1'/x_1,\ldots,v_k'/x_k]\lceil e_0\rceil_{\widetilde{v'}} \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}_0}} e_2$$

for some e_2 such that $E_2[f\,w_1\,\cdots\,w_k] \sim_{\widetilde{v'}} e_2$. Because $\mathit{arity}(f) = k$, e_2 must be of the form $E_2'[f\,s_1\,w_1'\,\cdots\,\widetilde{v'}\,w_k']$, and the required conditions are satisfied. □

Proof. **[Theorem 3]** By the assumption and Lemma 3, either we have $\texttt{main}\,\bot\,() \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}} E[f\,s_1\,w_1'\,\cdots\,s_k\,w_k']$ for some E,$s_1,\ldots,s_k$ and $w_1',\ldots,w_k'$ such that $s_k = \langle v_1',\ldots,v_k'\rangle$ with $\lfloor v_i'\rfloor = v_i$ and $\lfloor w_i'\rfloor = w_i$ for $i \in \{1,\ldots,k\}$, or $\texttt{main}\,\bot\,() \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}}$ $\texttt{fail}$ (because $\rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}}$ is the same except that the assertions are stronger). In the former case, by the assumption

$$(\langle v_1,\ldots,v_k\rangle,\langle w_1,\ldots,w_k\rangle) \notin \mathcal{D},$$

we have $\texttt{main}\,\bot\,() \rightarrow^*_{\lceil P\rceil_{f,\mathcal{D}}}$ $\texttt{fail}$ as required. □

D Proof of Theorem 4

Lemma 4. *Let P be a program, e_1 be a closed expression, and s be $\perp$ or a tuple of values $\langle v\rangle$. If $e'_1 \rightarrow^{+}_{\lceil P\rceil_{f,\mathcal{D}_0}} e'_2$, then $\lfloor e'_1\rfloor \rightarrow_P e_2$ and either $e'_2 \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} e''_2$ or $e''_2 \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} e'_2$ for some e_2 and e''_2 such that $e_2 = \lfloor e''_2\rfloor$.*

Proof. This follows by straightforward induction on the case analysis on the rule used for the first reduction step of $e'_1 \rightarrow^{+}_{\lceil P\rceil_{f,\mathcal{D}_0}} e'_2$. The only non-trivial case is when

$$\begin{array}{l} e'_1 \equiv E'[f\, s_1\, v'_1 \cdots s_k\, v'_k] \rightarrow_{\lceil P\rceil_{f,\mathcal{D}_0}} \\ \texttt{let}\ s = \mathbf{check\&upd}(\mathcal{D}^{\#}_0, s_k, \langle v'_1,\ldots,v'_k\rangle)\ \texttt{in} \\ \quad [v'_1/x_1,\ldots,v'_k/x_k]\lceil e_0\rceil_s \equiv e'_2 \end{array}$$

with $f\ x_1\ \cdots\ x_k = e_0 \in P$. In this case, we have $e_1 \equiv E[f\,\widetilde{v}]$ with $E = \lfloor E'\rfloor$ and $\lfloor\widetilde{v}\rfloor = \lfloor\widetilde{v}'\rfloor$. The required result holds for $e_2 = [v_1/x_1,\ldots,v_k/x_k]e_0$ and $e''_2 = E'[[v'_1/x_1,\ldots,v'_k/x_k]\lceil e_0\rceil_{s'}]$ where s' is s_k or $\langle v'_1,\ldots,v'_k\rangle$. $\square$

Lemma 5. *If $\texttt{main}\perp() \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} E[g\,s_1\,w'_1 \cdots s_\ell\,w'_\ell]$ with $s_\ell = \langle v'_1,\ldots,v'_k\rangle$ and $arity(g) = \ell$, then $\texttt{main}\,() \rightarrow^{*}_P E_1[f\,v_1 \cdots v_k]$ and $f\,v_1 \cdots v_k \rightarrow^{+}_P E_2[g\,w_1 \cdots w_\ell]$, for some $E_1, E_2, \widetilde{v}, \widetilde{w}$ such that $\lfloor v'_i\rfloor = v_i$ for $i \in \{1,\ldots,k\}$. and $\lfloor w'_i\rfloor = w_i$ for $i \in \{1,\ldots,\ell\}$.*

Proof. Suppose $\texttt{main}\perp() \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} E[g\,s_1\,w'_1 \cdots s_k\,w'_\ell]$ and $s_\ell = \langle v'_1,\ldots,v'_k\rangle$. Then, by the definition of the transformation, it must be the case that $\texttt{main}\perp() \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} E'_1[f\,s'_1\,v'_1 \cdots s'_k\,v'_k]$ and $f\,s'_1\,v'_1 \cdots s'_k\,v'_k \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} E'_2[g\,s_1\,w'_1 \cdots s_\ell\,w'_\ell]$ with $E = E'_1[E'_2]$. (Formally, this is proved by induction on the length of the reduction sequence.) By Lemma 4, we obtain the required property. $\square$

Proof. **[Theorem 4]** By the assumption $\texttt{main}\perp() \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}}} \texttt{fail}$, we have

$$\texttt{main}\perp() \rightarrow^{*}_{\lceil P\rceil_{f,\mathcal{D}_0}} E[f\,s_1\,w'_1 \cdots s_k\,w'_k]$$

with $(s_k, w'_k) \notin \mathcal{D}^{\#}$. Thus, the statement follows from Lemma 5. $\square$

E Implicit Parameter Instantiation Inference Example

Consider the program `indirectHO` from Section 3.4. As remarked there, an implicit parameter is added to the program as follows.

```
let app h_IMPARAM h v = h () v
let id x = x
let rec g x u =
      if x <= 0 then id else app σ(ℓ) (g (x-1))
let main () = g *int () ()
```

We initialize $\sigma(\ell) = 0$.

The transformed program $\lceil\texttt{indirectHO}\rceil_{\sigma,\texttt{app},\emptyset}$ is shown below.[13]

[13] Here, the program is simplified and only has the implicit parameter is tracked via the extra parameter for carrying call-tree ancestor's arguments. That is, it only tracks the previous `h_IMPARAM` (in `ph_IMPARAM`) and does not track the previous values of `h` and `v`.

```
let app _ h_IMPARAM _ h ph_IMPARAM v =
  assert (ph_IMPARAM ≠ ⊥);
  let s = if *_bool then ph_IMPARAM else h_IMPARAM in
  h s () s v
let id s x = x
let rec g _ x s u =
  if x ≤ 0 then id
  else app s 0 s (g s (x-1))
let main xs () = g xs *_int xs ()
```

The program causes an assertion failure, and we obtain the counterexample as an error path in $\lceil\texttt{indirectHO}\rceil_{\texttt{app},\emptyset}$. Suppose we obtained the error path represented as the following reduction sequence:

$$\begin{aligned}\texttt{main}\perp() &\to^{*}\texttt{g}\perp\texttt{2}\perp()\\ &\to^{*}\texttt{app}\perp\sigma(\ell)\perp(\texttt{g}\perp\texttt{1})\perp()\\ &\to^{*}\texttt{g}\,\sigma(\ell)\,\texttt{1}\,\sigma(\ell)\,()\\ &\to^{*}\texttt{app}\,\sigma(\ell)\,\sigma(\ell)\,\sigma(\ell)\,(\texttt{g}\,\sigma(\ell)\,\texttt{0})\,\sigma(\ell)\,()\\ &\to^{*}\texttt{assert}(\sigma(\ell){=}\perp)\to\texttt{fail}\end{aligned}$$

To build the strongest post condition of the counterexample, we build the *straight-line higher-order program* (henceforth abbreviated to *SHP*) corresponding to the counterexample path. SHP is a recursion-free slice of the program obtained by copying functions and removing branches so that it contains no branches and every function occurrence is "linear" (i.e., each function is called at most once), such that the evaluating the SHP results in the counterexample path as the reduction sequence. We refer to the previous work [16, 33] for more details on SHP.

From the counterexample above, we obtain the following SHP π.[14] (Here, the expression "`assume(`*cond*`);`e" evaluates e if *cond* is true and otherwise gets stuck safely.)

```
let rec main xs () = g^(1) xs *_int xs ()
and g^(1) _ x s u =
  assume (¬(x ≤ 0)); app^(1) s ℓ_g(1) s (g^(2) s (x-1))
and app^(1) _ h_IMPARAM _ h ph_IMPARAM v =
  assume (¬(ph_IMPARAM ≠ ⊥));
  let s = h_IMPARAM in h s () s v
and g^(2) _ x s u =
  assume (¬(x ≤ 0)); app^(2) s ℓ_g(2) s (g^(3) s (x-1))
and g^(3) _ x s () = assume (false); u
and app^(2) _ h_IMPARAM _ h ph_IMPARAM () =
  assume (ph_IMPARAM≠ ⊥); fail
```

[14] Unlike in the main body of the paper, here we use π to denote the SHP instead of the counterexample path.

We prepare the template $c_{1,0} + c_{1,1}\mathtt{x}_{\mathtt{g}^{(1)}}$ (resp. $c_{2,0} + c_{2,1}\mathtt{x}_{\mathtt{g}^{(2)}}$) for the instantiation site $\ell_{\mathtt{g}^{(1)}}$ (resp. $\ell_{\mathtt{g}^{(2)}}$) with fresh unknowns. (Generally, for an instantiation site ℓ, we prepare the template $c_0 + c_1 x_1 + \cdots + c_m x_m$ where $x_1, \ldots, x_m$ are the variables bound in the context of ℓ.) We let $\Delta = \left[c_{1,0} + c_{1,1}\mathtt{x}_{\ell_{\mathtt{g}^{(1)}}} / \ell_{\mathtt{g}^{(1)}}\right] \left[c_{2,0} + c_{2,1}\mathtt{x}_{\ell_{\mathtt{g}^{(1)}}} / \ell_{\mathtt{g}^{(2)}}\right]$ be the template instantiation map.

Then, the following strongest postcondition $[\![\pi\Delta]\!]$ is computed by symbolically evaluating $\pi\Delta$. (The formula is slightly simplified.)

$$\begin{aligned}[\![\pi\Delta]\!] \equiv\ & \mathtt{x}_{\mathtt{g}^{(1)}} > 0 \land \mathtt{h\_IMPARAM}_{\mathsf{app}^{(1)}} = \sigma(\ell_{\mathtt{g}^{(1)}}) \\ & \land \mathtt{ph\_IMPARAM}_{\mathsf{app}^{(1)}} = \bot \land \mathtt{x}_{\mathtt{g}^{(2)}} = \mathtt{x}_{\mathtt{g}^{(1)}} - 1 \land \mathtt{x}_{\mathtt{g}^{(2)}} > 0 \\ & \land \mathtt{h\_IMPARAM}_{\mathsf{app}^{(2)}} = \sigma(\ell_{\mathtt{g}^{(2)}}) \\ & \land \mathtt{ph\_IMPARAM}_{\mathsf{app}^{(2)}} = \mathtt{h\_IMPARAM}_{\mathsf{app}^{(1)}} \\ & \land \mathtt{ph\_IMPARAM}_{\mathsf{app}^{(2)}} \neq \bot\end{aligned}$$

Next, we prepare the ranking function template $T(x) = c_0 + c_1 x$ with fresh unknowns and solve for the assignments to the unknowns satisfying the constraint below.

$$\forall \widetilde{x}. [\![\pi\Delta]\!] \Rightarrow T(\mathtt{ph\_IMPARAM}_{\mathsf{app}^{(2)}}) > T(\mathtt{h\_IMPARAM}_{\mathsf{app}^{(2)}}) \geq 0$$

Suppose we obtain the following as the solution to the constraint.

$$c_0 \mapsto 0, c_1 \mapsto 1, c_{1,0} \mapsto 0, c_{1,1} \mapsto 1, c_{2,0} \mapsto 0, c_{2,1} \mapsto 1$$

This gives us the new ranking function $r(x) = T(x)\,[0/c_0]\,[1/c_1]$ and the new implicit parameter instantiation $\sigma' = \Delta\,[0/c_{1,0}]\,[1/c_{1,1}]\,[0/c_{2,0}]\,[1/c_{2,1}]$. We refine $\mathcal{D}_{\mathsf{app}}$ from $\emptyset$ to $\{(x, x') \mid r(x) > r(x') \geq 0\}$, and the implicit parameter instantiation to σ'.

With the refined candidate well-founded relation and implicit parameter instantiation, the reachability checker is able to verify that $\lceil \mathtt{indirectHO} \rceil_{\sigma', \mathsf{app}, \mathcal{D}_{\mathsf{app}}}$ is assertion safe. And, the program is verified to be terminating.

F Lexicographic Linear Ranking Function

Cook et al. [5] have proposed a counterexample-guided method to infer a class of well-founded relations called *lexicographic linear ranking functions* (LLRFs). LLRFs is a sequence of ranking functions $\langle r_1, \ldots, r_n \rangle$ that represents the following well-founded relation.

$$\{(\widetilde{x}', \widetilde{x}) \mid \bigvee_{i \leq n} r_i(\widetilde{x}') > r_i(\widetilde{x}) \geq 0 \land \bigwedge_{j \leq i-1} r_j(\widetilde{x}') \leq r_j(\widetilde{x})\}$$

Inferring LLRFs can be done by a constraint-based method similar to the ranking function inference described in Section 3.4.

As observed in [5], one can often find LLRFs that refutes all the currently-seen counterexamples to form a sufficient termination argument, which, in turn, allows the verification process to check the termination argument against the call tree relation as opposed to the *transitive closure* of the call tree relation. In an approach based on a reduction to binary reachability, this means that we can use an optimized version of the program transformation that avoids building the transitive closure. We informally describe the optimized transformation using an example.

Example 6. Consider the following program.

```
let rec f m n =
  let r = *int in
    if r > 0 && m > 0 then f (m-1) *int
    else if r <= 0 && n > 0 then f m (n-1)
    else ()
let main () = f *int *int
```

Note that either `m` or `n` is decremented non-deterministically by a recursive call to `f`. Note that we cannot verify the program's termination via one linear ranking function, because the recursion `f m n`$\triangleright$`f (m-1)` $*_{\mathbf{int}}$ only depends on the argument `m` while the recursion `f m n`$\triangleright$`f m (n-1)` is not strictly decreasing in `m`.

Nonetheless, we can verify the program's termination by using $\langle \mathtt{m}, \mathtt{n} \rangle$ as LLRFs of `f`. To do this, we transform the program as follows.

```
let rec f m (pm, pn) n =
  assert (( pm > m && m >= 0)
       || ( pm >= m && pn > n && n >= 0))
  let (pm, pn) = (m, n) in
  let r = *int in
    if r > 0 && m > 0 then f (m-1) (pm, pn) n
    else if r <= 0 && n > 0 then f m (pm, pn) (n-1)
    else ()
let main s () = f *int s *int
```

The underlines highlight the main differences from the transformation described in the main body of the paper. Firstly, the "not decreasing" condition `pm >= m` of the first ranking function `m` is added as the condition guarding the second ranking function (i.e., `pn > n && n >= 0`). Secondly, the state `(pm, pn)` is *deterministically* updated at the beginning the body of `f`. The reachability checker will find that the program is assertion safe, and we have verified the program's termination.

G Further Description of Benchmark Programs

- `indirectIntro` is P_1 from Section 1. That is, it is `indirect` from Example 2 and 4 but without the simplification. We show the code below.

  ```
  let rec app f x u =
    if x>0 then app f (x-1) u else f x u
  let id u = ()
  let rec g x = if x <= 0 then id else app g x
  let main () = g *int ()
  ```

- `indirect` is from Example 2 and 4.
- `indirectHO` is from Section 3.4.
- `CE-0CFA` is the program below.

```
let id x = x
let rec omega x = omega x
let f x y z = y z
let main () = f (f id omega) id 1
```

The program is from [29], given as an example on which both 0CFA and tree-automata-based size-change analysis fails. `CE-1CFA` is its variant obtained by wrapping each function with an apply function so that 1-limited CFA [29] also fails. See also the discussion in Section 5.

– `up_down` is the program below.

```
let rec app f x = f x
and down x = if x = 0 then () else down (x-1)
and up x = if x = 0 then () else up (x+1)
let main () =
  let t1 = *int in let t2 = *int in
  if t1 > 0 then app down t1
  else if t2 < 0 then app up t2 else ()
```

The program cannot be verified by 0CFA-based size-change analysis because `up` and `down` are both passed to the first argument of `f` and the analysis cannot distinguish the two functions.

– `churchNum` is the program below, which is based on the example given in Section 7.1 of [12].

```
let succ m s z = m s (s z)
let id x = x
let two f z = f (f z)
let zero f z = z
let main () = two succ zero id 0
```

– `CE-Jones_Bohr` is the following program.

```
let f1 u c d = d
let f2 u a b = a (f1 u)
let f3 u a = a (f2 u a)
let f4 u v = v
let f5 u e = e (f4 u)
let main () = let zz = f3 u (f5 u) in ()
```

The program is the λ-lifting of $(\lambda a.a(\lambda b.a(\lambda cd.d)))(\lambda e.e(\lambda f.f))$, which is given in Section 7.8 of [12] and is used to show a limitation of their approach.

– `map` is the program below, which is based on the example given in Section 3.1.1 of [28].

```
let rec map f xs =
  if xs = 0 then 0
  else f *int + map f (xs - 1)
```

```
let compose f g x = f (g x)
let add x y = x + y
let main () =
  let l = *int in
  if l >= 0 then map (compose (add 1) (add 2)) l else 0
```

– `toChurch` is the program below.

```
let compose f g x = f (g x)
let id x = x
let succ x = x + 1
let rec toChurch n f =
  if n = 0 then id
  else compose f (toChurch (n - 1) f)
let main () =
  let x = *int in
  if x>=0 then let tos = toChurch x succ in ()
  else ()
```

The program is given in Section 4.1 of [28] as an example whose dynamic call graph includes closures of unbounded depth.

– `x_plus_2^n` is the following program given in [12], Section 7.2.

```
let succ n = n + 1
let g r a = r (r a)
let rec f n = if n=0 then succ else g (f (n-1))
let main () =
  let n = *int in
  let x = *int in
  if n>=0 && x>=0 then f n x else 0
```

– `foldr` is the program below, which is based on the example given in Section 4.4.4 of [28].

```
let rec foldr h e l =
  if l = 0 then e
  else h *int (foldr h e (l-1))
let sum m n = m + n
let main () =
  let l = *int in
  if l >= 0 then foldr sum *int l else 0
```