Predicate Abstraction and CEGAR for Disproving Termination of Higher-order Functional Programs

Takuya Kuwahara[1], Ryosuke Sato[2], Hiroshi Unno[3], and Naoki Kobayashi[2]

[1] Knowledge Discovery Research Laboratories, NEC, kuwahara@me.jp.nec.com
[2] The University of Tokyo, {ryosuke,koba}@kb.is.s.u-tokyo.ac.jp
[3] University of Tsukuba, uhiro@cs.tsukuba.ac.jp

Abstract. We propose an automated method for disproving termination of higher-order functional programs. Our method combines higher-order model checking with predicate abstraction and CEGAR. Our predicate abstraction is novel in that it computes a mixture of under- and overapproximations. For non-determinism of a source program (such as random number generation), we apply underapproximation to generate a subset of the actual branches, and check that some of the branches in the abstract program is non-terminating. For operations on infinite data domains (such as integers), we apply overapproximation to generate a superset of the actual branches, and check that every branch is non-terminating. Thus, disproving non-termination reduces to the problem of checking a certain branching property of the abstract program, which can be solved by higher-order model checking. We have implemented a prototype non-termination prover based on our method and have confirmed the effectiveness of the proposed approach through experiments.

1 Introduction

We propose an automated method for disproving termination of higher-order functional programs (i.e., for proving that a given program does not terminate for *some* input). The method plays a role complementary to the automated method for proving termination of higher-order programs (i.e., for proving that a given program terminates for *all* inputs) [17]. Several methods have recently been proposed for proving non-termination of programs [7–9, 11, 13, 14, 18], but most of them have focused on *first-order* programs (or, while programs) that can be represented as finite control graphs. An exception is work on term rewriting systems (TRS) [9, 11]; higher-order programs can be encoded into term rewriting systems, but the definition of non-termination is different: TRS is non-terminating if there exists a term that has a non-terminating rewriting sequence, not necessarily the initial term.

Our approach is based on a combination of higher-order model checking [15, 20] with predicate abstraction and CEGAR (counterexample-guided abstraction refinement). Values of a base type (such as integers) are abstracted to (tuples of) Booleans by using predicates, and higher-order functions are abstracted accordingly. Higher-order model checking is then used to analyze the abstracted program. A combination of predicate abstraction and higher-order model checking has been previously proposed for verifying safety properties of higher-order programs (i.e., for proving that a program does not reach an error state in *all* execution paths) [16]. With respect to that work, the approach of the present paper is novel in that we combine *over*approximation and *under*approximation. Note that predicate abstraction [3, 12, 16] usually yields an overapproximation, i.e., an abstract program that contains a *superset* of possible execution paths of the original program. With such an abstraction, non-termination of the abstract program (the existence of a non-terminating path) does not imply that of the original program. To address this problem, we use both under- and overapproximations. For a deterministic computation step of the original program, we apply overapproximation but check that *every* branch of the overapproximation has a non-terminating path. For a non-deterministic branch of the original program (such as random number generation and an input from the environment), we apply *under*-approximation, and check that *some* branch of the underapproximation has a non-terminating path.

Figure 1 illustrates how under- and overapproximations are combined. The program considered here is of the form:

$$\textbf{let } x = * \textbf{ in let } y = x+1 \textbf{ in let } z = * \textbf{ in } \cdots .$$

Here, $*$ generates a random integer. Thus, the program has the execution tree shown on the top of Figure 1. The first and third steps are non-deterministic, while the second step (corresponding to $y = x+1$) is deterministic. Suppose that the predicates used for abstracting the values of x, y, and z are $x > 0$, $y > 0$, and $0 \leq z < x+y$ (these predicates do not necessarily yield a good abstraction, but are sufficient for explaining the combination of under- and overapproximations). Then, the abstract program has the execution tree shown on the bottom of the figure. Due to the predicate abstraction, the infinitely many branches on the value of x have been replaced by two branches $x > 0$ and $\neg x > 0$. The node $\exists$ means that only one of the branches needs to have an infinite path (for the original program having a non-terminating path). The deterministic path from $x = n$ to $y = n+1$ has now been replaced by non-deterministic branches $y > 0$ and $\neg y > 0$. The node $\forall$ indicates that *every* child of the node must have an infinite path. Below the node $x > 0$, however, we do not have a node for $\neg y > 0$, as $x > 0$ and $y = x+1$ imply $y > 0$. The infinite branches on z have been replaced by non-deterministic branches on $\neg(0 \leq z < x+y)$ or $0 \leq z < x+y$. As is the case for x, the branches are marked by $\exists$, meaning that one of the branches needs to have an infinite path. Note that below the node $\neg x > 0$, we only have a branch for $\neg(0 \leq z < x+y)$. This is because, when $x \leq 0$, there may be no z that satisfies $0 \leq z < x+y$; so, even if there may be an infinite execution sequence along that path, we cannot conclude that the source program is non-terminating. Thus, this part of the tree provides an *under*-approximation of the source program.

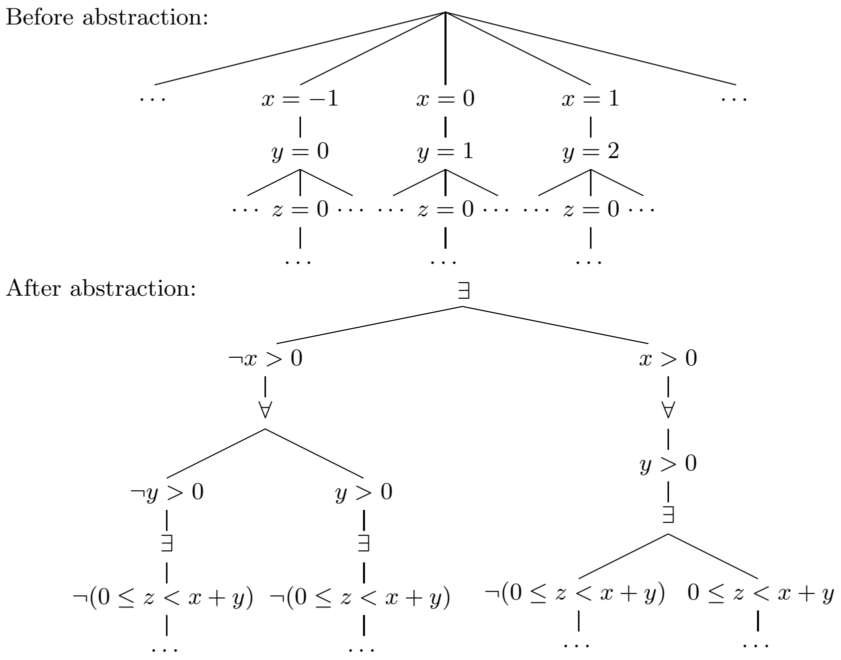


Fig. 1. Execution trees before/after abstraction

An abstract program is actually represented as a tree-generating program that generates an execution tree like the one shown on the bottom of Figure 1. Higher-order model checking is then used for checking, informally speaking, that *every* child of each $\forall$-node has a non-terminating path, and that *some* child of each $\exists$-node has a non-terminating path.

The use of overapproximation for disproving termination has also been proposed recently by Cook et al. [8]. Although their theoretical framework is general, their concrete method for automation is limited to first-order programs. They also propose a restricted form of combination of underapproximation and overapproximation, but underapproximation can be followed by overapproximation, but not vice versa.

The rest of this paper is structured as follows. Section 2 defines the language used as the target of our verification. Sections 3 and 4 describe predicate abstraction and CEGAR respectively. Section 5 reports experiments. Section 6 discusses related work and Section 7 concludes the paper.

2 Language

In this section, we introduce the language of source programs, used as the target of non-termination verification. It is a simply typed, call-by-value higher-order functional language. Throughout the paper, we often use the following abbrevi-

$$
\begin{array}{l}
P \text{ (programs)} := \{f_i\ \widetilde{x}_i = e_i\}_{i\in\{1\dots n\}} \\
e \text{ (expressions)} := (\,) \mid y\,\widetilde{v} \mid \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
\qquad\qquad\qquad\quad \mid \texttt{let}\ x\,\texttt{=}\,a\ \texttt{in}\ e \mid \texttt{let}\ x\,\texttt{=}\,\texttt{*}_{\texttt{int}}\ \texttt{in}\ e \\
a \text{ (simple expressions)} ::= x \mid n \mid \mathrm{op}\,(\widetilde{a}) \qquad\qquad v \text{ (values)} := n \mid y\ \widetilde{v}
\end{array}
$$

$$
\frac{f\ \widetilde{x} = e \in P \quad |\widetilde{x}| = |\widetilde{v}|}{f\ \widetilde{v} \longrightarrow_P [\widetilde{v}/\widetilde{x}]\, e}
\qquad
\frac{[\![a]\!] = n}{\texttt{let}\ x\,\texttt{=}\,a\ \texttt{in}\ e \longrightarrow_P [n/x]\, e}
\qquad
\texttt{let}\ x\,\texttt{=}\,\texttt{*}_{\texttt{int}}\ \texttt{in}\ e \longrightarrow_P [n/x]\, e
$$

$$
\texttt{if}\ n\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \longrightarrow_P e_0 \text{ if } n \neq 0 \qquad \texttt{if}\ 0\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \longrightarrow_P e_1
$$

Fig. 2. The syntax and operational semantics of the language

ations: $\widetilde{e}$ for a possibly empty sequence $e_1, \dots, e_n$, and $\{e_i\}_{i\in\{1,\dots,n\}}$ for the set $\{e_1, \dots, e_n\}$.

The syntax and operational semantics of the language is given in Figure 2. The meta-variable f_i ranges over a set of function names, and x, y range over the set of function names and ordinary variables. The meta-variable n ranges over the set of integers, and op over a set of integer operations. We omit Booleans and regard a non-negative integer as true, and 0 as false. We require that $y\,\widetilde{v}$ in the definition of e is a full application, i.e., that all the necessary arguments are passed to y, and $y\,\widetilde{v}$ has a base type. In contrast, $y\,\widetilde{v}$ in the definition of v is a partial application. Whether $y\,\widetilde{v}$ is a full or partial application is actually determined by the simple type system mentioned below.

The expression $\texttt{let}\ x\,\texttt{=}\,\texttt{*}_{\texttt{int}}\ \texttt{in}\ e$ randomly generates an integer, binds x to it, and evaluates e. The meanings of the other expressions should be clear. A careful reader may notice that we have only tail calls. This is for the simplicity of the presentation. Note that it does not lose generality, because we can apply the standard continuation-passing-style (CPS) transformation to guarantee that all the function calls are in this form. We assume that for every program $\{f_i\ \widetilde{x}_i = e_i\}_{i\in\{1\dots n\}}$, $\texttt{main} \in \{f_1, \dots, f_n\}$.

We assume that programs are simply-typed. The syntax of types is given by: $\tau ::= \mathbf{int} \mid \star \mid \tau_1 \to \tau_2$. The types **int** and $\star$ describe integers and the unit value $(\,)$ respectively. The type $\tau_1 \to \tau_2$ describes functions from τ_1 to τ_2. The typing rules for expressions and programs are deferred Appendix A, which are standard except that the body of each function definition can only have type $\star$. This does not lose generality since the CPS transformation guarantees this condition.

The one-step reduction relation $e_1 \longrightarrow_P e_2$ is defined by the rules in Fig. 2, where $[\![a]\!]$ stands for the value of the simple expression a. A program P is *non-terminating* if there is an infinite reduction sequence $\texttt{main} \to_P e_1 \to_P e_2 \to_P \dots$.

3 Predicate Abstraction

3.1 Target Language

The target language of predicate abstraction is a higher-order simply-typed functional language having Booleans and special tree constructors as primitives. The syntax is given in Figure 3. We assume that for every program

$$\begin{array}{l}
D \text{ (programs)} := \{f_i\ \widetilde{x}_i = M_i\}_{i\in\{1\ldots n\}} \\
M \text{ (expressions)} := c(M_1,\ldots,M_k) \mid y\,\widetilde{V}\ \mid \texttt{let}\ x\,\texttt{=}\,(b_1,\ldots,b_k)\ \texttt{in}\ M \\
\qquad\qquad\qquad \mid \texttt{br}_\forall\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\} \mid \texttt{br}_\exists\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\} \\
b \text{ (Booleans)} ::= \texttt{true} \mid \texttt{false} \qquad V \text{ (values)} ::= (b_1,\ldots,b_k) \mid y\,\widetilde{V} \\
\psi \text{ (Boolean expressions) } ::= b \mid \#_i(x) \mid \psi_1 \vee \psi_2 \mid \neg\psi
\end{array}$$

$$E[f\,V_1\cdots V_k] \longrightarrow_D E[[V_1/x_1,\ldots,V_k/x_k]M] \text{ if } f\,x_1\cdots x_k = M \in D$$

$$E[\texttt{let}\ x\,\texttt{=}\,(b_1,\ldots,b_k)\ \texttt{in}\ M] \longrightarrow_D E[[(b_1,\ldots,b_k)/x]M]$$

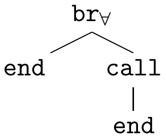
$$\begin{array}{l}
E[\texttt{br}_\forall\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\}] \longrightarrow_D E[\texttt{br}_\forall(M_{i_1},\ldots,M_{i_\ell})] \\
\qquad\qquad\qquad \text{if } \{\psi_i \mid i \in \{1,\ldots,k\}\,, [\![\psi_i]\!] = \texttt{true}\} = \{\psi_{i_1},\ldots,\psi_{i_\ell}\}
\end{array}$$

$$\begin{array}{l}
E[\texttt{br}_\exists\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\}] \longrightarrow_D E[\texttt{br}_\exists(M_{i_1},\ldots,M_{i_\ell})] \\
\qquad\qquad\qquad \text{if } \{\psi_i \mid i \in \{1,\ldots,k\}\,, [\![\psi_i]\!] = \texttt{true}\} = \{\psi_{i_1},\ldots,\psi_{i_\ell}\}
\end{array}$$

$$E \text{ (evaluation contexts) } ::= [\,] \mid c(M_1,\ldots,M_{i-1},E,M_{i+1},\ldots,M_n)$$

Fig. 3. The syntax and semantics of the target language

$\{f_i\ \widetilde{x}_i = M_i\}_{i\in\{1\ldots n\}}$, $\texttt{main} \in \{f_1,\ldots,f_n\}$. Each expression generates a possibly infinite tree, describing possible executions of a source program. The expression $c(M_1,\ldots,M_k)$ generates a node labeled with c, having the trees generated by $M_1,\ldots,M_k$ as its children. Here, c ranges over the set $\{\texttt{end},\texttt{call},\texttt{br}_\forall,\texttt{br}_\exists\}$ of tree constructors. The constructors `end` and `call` have arities 0 and 1 respectively, while $\texttt{br}_\forall$ and $\texttt{br}_\exists$ may have arbitrarily many children. We just write `end` for `end()`. The expression $\texttt{let}\ x\,\texttt{=}\,(b_1,\ldots,b_k)\ \texttt{in}\ M$ binds x to the tuple $(b_1,\ldots,b_k)$, and evaluates M. The expression $\texttt{br}_\forall\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\}$ ($\texttt{br}_\exists\,\{\psi_1 \to M_1,\ldots,\psi_k \to M_k\}$, resp.) generates the node $\texttt{br}_\forall$ ($\texttt{br}_\exists$, resp.), and adds the tree generated by M_i as a child of the node if ψ_i evaluates to `true`, where the order of children does not matter. The Boolean expression $\#_i x$ denotes the i-th element of the tuple x. For example, $\texttt{let}\ x\,\texttt{=}\,(\texttt{true},\texttt{false})\ \texttt{in}\ \texttt{br}_\forall\{\#_1(x) \to \texttt{end}, \#_2(x) \to \texttt{call}(\texttt{call}(\texttt{end})), \#_1(x)\vee\#_2(x) \to \texttt{call}(\texttt{end})\}$ generates the tree:



The formal semantics is given through the reduction relation $M \longrightarrow_D M'$, defined in Figure 3. The tree generated by a program D, written $\mathbf{Tree}(D)$, is the "limit" of the trees obtained from a (possibly infinite) reduction sequence $\texttt{main} \longrightarrow_D M_1 \longrightarrow_D M_2 \longrightarrow_D \cdots$. For example, the program $\{\texttt{main} = \texttt{call}(\texttt{main})\}$ generates an infinite (linear) tree consisting of infinitely many `call` nodes.

Intuitively, the tree generated by a program of the target language describes possible execution sequences of a source program. The property that a source program has a non-terminating execution sequence is transformed to the property of the tree that (i) every child of each $\mathtt{br}_\forall$ node has an infinite path, and (ii) some child of each $\mathtt{br}_\exists$ node has an infinite path. More formally, the set of (infinite) trees that represent the existence of a non-terminating computation is the largest set **NonTermTrees** such that for every $T \in \mathbf{NonTermTrees}$, T satisfies one of the following conditions.

1. $T = \mathtt{call}(T')$ and $T' \in \mathbf{NonTermTrees}$
2. $T = \mathtt{br}_\forall(T_1, \dots, T_k)$ and $T_i \in \mathbf{NonTermTrees}$ for all $i \in \{1, \dots, k\}$.
3. $T = \mathtt{br}_\exists(T_1, \dots, T_k)$ and $T_i \in \mathbf{NonTermTrees}$ for some $i \in \{1, \dots, k\}$.

The property above can be expressed by MSO (the monadic second order logic; or equivalently, modal μ-calculus or alternating parity tree automata); thus whether the tree generated by a program of the target language belongs to **NonTermTrees** can be decided by higher-order model checking [15, 20].

3.2 Abstraction

We now formalize predicate abstraction for transforming a source program to a program (of the target language) that generates a tree that approximately represents the possible execution sequences of the source program. Following Kobayashi et al. [16], we use *abstraction types* for expressing which predicate should be used for abstracting each value. The syntax of abstraction types is:

$$
\begin{array}{l}
\sigma \text{ (abstraction types)} ::= \star \mid \mathbf{int}[Q_1, \dots, Q_k] \mid x : \sigma_1 \to \sigma_2 \\
Q \text{ (predicates)} ::= \lambda x.\varphi \qquad \varphi ::= n_1 x_1 + \cdots + n_k x_k \leq n \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi
\end{array}
$$

The type $\star$ describes the unit value, and $\mathbf{int}[Q_1, \dots, Q_k]$ describes an integer that should be abstracted by using the predicates $Q_1, \dots, Q_k$. For example, given an abstraction type $\mathbf{int}[\lambda x.x \leq 1, \lambda x.2x - 1 \leq 0]$, the integer 1 is abstracted to $(\mathtt{true}, \mathtt{false})$. In the syntax above, we list only linear inequalities as primitive constraints, but we can include other constraints (such as those on uninterpreted function symbols) as long as the underlying theory remains decidable. The type $x : \sigma_1 \to \sigma_2$ describes a function whose argument and return value should be abstracted according to σ_1 and σ_2 respectively. In σ_2, the argument can be referred to by x if x has an integer type $\mathbf{int}[Q_1, \dots, Q_k]$. For example, $x : \mathbf{int}[\lambda x.x \leq 0] \to \mathbf{int}[\lambda y.y - x \leq 0]$ describes a function from integers to integers whose argument should be abstracted using the predicate $\lambda x.x \leq 0$ and whose return value should be abstracted using $\lambda y.y - x \leq 0$. Thus, the successor function (defined by $f\,x = x + 1$) will be abstracted to a Boolean function $\lambda b.\mathtt{false}$ (because the return value $x + 1$ is always greater than x, no matter whether $x \leq 0$ or not).

The predicate abstraction for expressions and programs is formalized using the relations $\mathit{\Gamma} \vdash e : \sigma \rightsquigarrow M$ and $\vdash P : \mathit{\Gamma} \rightsquigarrow D$, where $\mathit{\Gamma}$, called an *abstraction type environment*, is of the form $x_1 : \sigma_1, \dots, x_n : \sigma_n$. Intuitively, $\mathit{\Gamma} \vdash e : \sigma \rightsquigarrow M$ means that under the assumption that each free variable x_i of e is abstracted according to σ_i, the expression e is abstracted to M according to the abstraction

$$\frac{}{\mathit{\Gamma} \vdash (\,) : \star \rightsquigarrow \mathtt{end}} \quad \text{(PA-Unit)}$$

$$\frac{\begin{array}{c}\models b_1 Q_1(a) \wedge \cdots \wedge b_k Q_k(a) \Rightarrow \theta_{\mathit{\Gamma}} \psi_{(b_1,\ldots,b_k)} \ \text{(for each } b_1,\ldots,b_k \in \{\mathtt{true},\mathtt{false}\}) \\ \mathit{\Gamma}, x : \mathbf{int}[Q_1,\ldots,Q_k] \vdash e : \star \rightsquigarrow M\end{array}}{\begin{array}{c}\mathit{\Gamma} \vdash \mathtt{let}\ x : \mathbf{int}[Q_1,\ldots,Q_k] \mathtt{=} a\ \mathtt{in}\ e : \star \rightsquigarrow \\ \mathtt{br}_{\forall} \left\{\psi_{(b_1,\ldots,b_k)} \to \mathtt{let}\ x \mathtt{=} (b_1,\ldots,b_k)\ \mathtt{in}\ M \mid b_1,\ldots,b_k \in \{\mathtt{true},\mathtt{false}\}\right\}\end{array}}$$
$$\text{(PA-Sexp)}$$

$$\frac{\models x \neq 0 \Rightarrow \theta_{\mathit{\Gamma}} \psi_1 \qquad \models x = 0 \Rightarrow \theta_{\mathit{\Gamma}} \psi_2 \qquad \mathit{\Gamma} \vdash e_1 : \star \rightsquigarrow M_1 \qquad \mathit{\Gamma} \vdash e_2 : \star \rightsquigarrow M_2}{\mathit{\Gamma} \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \star \rightsquigarrow \mathtt{br}_{\forall} \{\psi_1 \to M_1, \psi_2 \to M_2\}}$$
$$\text{(PA-If)}$$

$$\frac{\begin{array}{c}\models \theta_{\mathit{\Gamma}} \psi_{(b_1,\ldots,b_k)} \Rightarrow \exists x. b_1 Q_1(x) \wedge \cdots \wedge b_k Q_k(x) \ \text{(for each } b_1,\ldots,b_k \in \{\mathtt{true},\mathtt{false}\}) \\ \mathit{\Gamma}, x : \mathbf{int}[Q_1,\ldots,Q_k] \vdash e : \star \rightsquigarrow M\end{array}}{\begin{array}{c}\mathit{\Gamma} \vdash \mathtt{let}\ x : \mathbf{int}[Q_1,\ldots,Q_k] \mathtt{=} \mathtt{*}_{\mathtt{int}}\ \mathtt{in}\ e : \star \rightsquigarrow \\ \mathtt{br}_{\exists} \left\{\psi_{(b_1,\ldots,b_k)} \to \mathtt{let}\ x \mathtt{=} (b_1,\ldots,b_k)\ \mathtt{in}\ M \mid b_1,\ldots,b_k \in \{\mathtt{true},\mathtt{false}\}\right\}\end{array}}$$
$$\text{(PA-Rand)}$$

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

$$\frac{\{f_i : \widetilde{x} : \widetilde{\sigma}_i \to \star\}_{i \in \{1,\ldots,k\}}, \widetilde{x} : \widetilde{\sigma}_j \vdash e_i : \star \rightsquigarrow M_i \ \text{for each } j \in \{1,\ldots,k\}}{\vdash \{f_i\, \widetilde{x}_i = e_i\}_{i \in \{1,\ldots,k\}} : \{f_i : \widetilde{x} : \widetilde{\sigma}_i \to \star\}_{i \in \{1,\ldots,k\}} \rightsquigarrow \{f_i\, \widetilde{x}_i = \mathtt{call}(M_i)\}_{i \in \{1,\ldots,k\}}}$$
$$\text{(PA-Prog)}$$

Fig. 4. Predicate Abstraction Rules

type σ. In the judgment $\vdash P : \mathit{\Gamma} \rightsquigarrow D$, $\mathit{\Gamma}$ describes how each function defined in P should be abstracted.

The relations are defined by the rules in Figure 4. Here, we consider, without loss of generality, only if-expressions of the form $\mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$. Also, function arguments are restricted to the syntax: $v ::= y\ \widetilde{v}$. (In other words, constants may not occur; note that $x\,c$ can be replaced by $\mathtt{let}\ y \mathtt{=} c\ \mathtt{in}\ x\,y$.) We assume that each let-expression is annotated with an abstraction type that should be used for abstracting the value of the variable. Those abstraction types, as well as those for functions are automatically inferred by the CEGAR procedure described in Section 4.

The rule PA-Unit just replaces the unit value with $\mathtt{end}$, which represents termination. The rule PA-Sexp overapproximates the value of a simple expression a. Here, $\theta_{\mathit{\Gamma}}$ is the substitution that replaces each variable x of type $\mathbf{int}[Q'_1,\ldots,Q'_n]$ in $\mathit{\Gamma}$ with $(Q'_1(x),\ldots,Q'_n(x))$. For example, if $\mathit{\Gamma} = x{:}\mathbf{int}[\lambda x.x \leq 0, \lambda x.x \leq 2], y : \mathbf{int}[\lambda y.y \leq x]$, then $\theta_{\mathit{\Gamma}}(\#_2(x) \wedge \#_1(y))$ is $\#_2(x \leq 0, x \leq 2) \wedge \#_1(y \leq x)$, i.e., $x \leq 2 \wedge y \leq x$. The formula $b_i Q_i(a)$ stands for $Q_i(a)$ if $b_i = \mathtt{true}$, and $\neg Q_i(a)$ if $b_i = \mathtt{false}$. Basically, the rule generates branches for all the possible values $(b_1,\ldots,b_k)$ for $(Q_1(a),\ldots,Q_k(a))$, and combines them with node $\mathtt{br}_{\forall}$ (which indicates that this branch has been obtained by an overapproximation). To eliminate impossible values, we compute a necessary condition $\psi_{(b_1,\ldots,b_k)}$ for $(Q_1(a),\ldots,Q_k(a)) = (b_1,\ldots,b_k)$ to hold, and guard the branch for

$(b_1,\dots,b_k)$ with $\psi_{(b_1,\dots,b_k)}$. The formula $\psi_{(b_1,\dots,b_k)}$ can be computed by using an SMT solver, as in ordinary predicate abstraction [3, 16]. (The rule generates 2^k branches, leading to code explosion. This is for the sake of simplicity; the eager splitting of branches is avoided in the actual implementation.) The rule PA-If is similar: branches for the then- and else-clauses are generated, but they are guarded by necessary conditions for the branches to be chosen.

The rule PA-Rand for random number generation is a kind of dual to PA-SExp. It applies an *underapproximation*, and generates branches for all the possible values $(b_1,\dots,b_k)$ for $(Q_1(x),\dots,Q_k(x))$ under the node $\mathtt{br}_\exists$. Each branch is guarded by a *sufficient* condition for the existence of a value for x such that $(Q_1(x),\dots,Q_k(x)) = (b_1,\dots,b_k)$, so that for each branch, there must be a corresponding execution path of the source program. The rule PA-App for applications is the same as the corresponding rule of [16]. Finally, the rule PA-Prog for programs just transforms the body of each function definition, but adds a special node `call` to keep track of function calls. Note that a program is non-terminating if and only if it makes infinitely many function calls.

Example 1 Let us consider the following program **LOOP**.

$$\begin{array}{l} \mathit{loop}\ h\ x = \mathtt{let}\ b = (x > 0)\ \mathtt{in} \\ \qquad\qquad \mathtt{if}\ b\ \mathtt{then}\ \mathtt{let}\ d = *_{\mathtt{int}}\ \mathtt{in}\ \mathtt{let}\ y = x + d\ \mathtt{in}\ h\ y\ (\mathit{loop}\ \mathit{app})\ \mathtt{else}\ (\,) \\ \mathit{app}\ m\ k = k\ m \qquad\qquad \mathtt{main} = \mathtt{let}\ r = *_{\mathtt{int}}\ \mathtt{in}\ \mathit{loop}\ \mathit{app}\ r \end{array}$$

LOOP is non-terminating; in fact, if $*_{\mathtt{int}}$ is always evaluated to 1, then we have:

$$\mathtt{main} \longrightarrow^* \mathit{loop}\ \mathit{app}\ 1 \longrightarrow^* \mathit{app}\ 2\ (\mathit{loop}\ \mathit{app}) \longrightarrow^* \mathit{loop}\ \mathit{app}\ 2 \longrightarrow^* \cdots$$

Let $\mathit{\Gamma}_{\mathbf{LOOP}}$ be an abstraction type environment:

$$\begin{array}{l} \mathit{loop} : (\mathbf{int}[\lambda\nu.\nu > 1] \to (\mathbf{int}[\lambda\nu.\nu > 1] \to \star) \to \star) \to \mathbf{int}[\lambda\nu.\nu > 1] \to \star \\ \mathit{app} : \mathbf{int}[\lambda\nu.\nu > 1] \to (\mathbf{int}[\lambda\nu.\nu > 1] \to \star) \to \star \end{array}$$

By using $\mathit{\Gamma}_{\mathbf{LOOP}}$ and the following abstraction types for b, d, and r:

$$b : \mathbf{int}[\lambda\nu.\nu \neq 0], d : \mathbf{int}[\lambda\nu.x + \nu > 1], r : \mathbf{int}[\lambda\nu.\nu > 1],$$

the program **LOOP** is abstracted to the following program $D_{\mathbf{LOOP}}$.

$$\begin{array}{l} \mathit{loop}\ h\ x = \mathtt{call}(\mathtt{br}_\forall\{\mathtt{true} \to \mathtt{let}\ b = \mathtt{true}\ \mathtt{in}\ M_1, \\ \qquad\qquad\qquad\qquad \neg x \to \mathtt{let}\ b = \mathtt{false}\ \mathtt{in}\ M_1\}) \\ \mathit{app}\ m\ k = \mathtt{call}(k\ m) \\ \mathtt{main} = \mathtt{call}(\mathtt{br}_\exists\{\mathtt{true} \to \mathtt{let}\ r = \mathtt{true}\ \mathtt{in}\ \mathit{loop}\ \mathit{app}\ r, \\ \qquad\qquad\qquad \mathtt{true} \to \mathtt{let}\ r = \mathtt{false}\ \mathtt{in}\ \mathit{loop}\ \mathit{app}\ r\}) \\ \text{where} \\ M_1 = \mathtt{br}_\forall\ \{b \to M_2, \neg b \to \mathtt{end}\} \\ M_2 = \mathtt{br}_\exists\ \{\mathtt{true} \to \mathtt{let}\ d = \mathtt{true}\ \mathtt{in}\ M_3, \mathtt{true} \to \mathtt{let}\ d = \mathtt{false}\ \mathtt{in}\ M_3\} \\ M_3 = \mathtt{br}_\forall\{d \to \mathtt{let}\ y = \mathtt{true}\ \mathtt{in}\ h\ y\ (\mathit{loop}\ \mathit{app}), \\ \qquad\quad \neg d \to \mathtt{let}\ y = \mathtt{false}\ \mathtt{in}\ h\ y\ (\mathit{loop}\ \mathit{app})\}. \end{array}$$

For example, $\texttt{let}\ b : \mathbf{int}[\lambda\nu.\nu \neq 0] \texttt{=} x > 0\ \texttt{in}\ e$ is transformed by PA-Sexp as follows:

$$\frac{\begin{array}{c}\models (x > 0) \neq 0 \Rightarrow \texttt{true} \qquad \models \neg((x > 0) \neq 0) \Rightarrow \neg(x > 1)(= \theta_{\Gamma}(\neg x)) \\ \Gamma, b : \mathbf{int}[\lambda\nu.\nu = 0] \vdash e : \star \rightsquigarrow M_1\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let}\ b : \mathbf{int}[\lambda\nu.\nu = 0] \texttt{=} x > 0\ \texttt{in}\ e \\ \rightsquigarrow \texttt{br}_{\forall}\,\{\texttt{true} \rightarrow \texttt{let}\ b \texttt{=} \texttt{true}\ \texttt{in}\ M_1, \neg x \rightarrow \texttt{let}\ b \texttt{=} \texttt{false}\ \texttt{in}\ M_1\}\end{array}}$$

where

$$\Gamma = \Gamma_{\mathbf{LOOP}}, h : (\mathbf{int}[\lambda\nu.\nu > 1] \rightarrow (\mathbf{int}[\lambda\nu.\nu > 1] \rightarrow \star) \rightarrow \star), x : \mathbf{int}[\lambda\nu.\nu > 1].$$

Here, recall that a non-zero integer is treated as `true` in the source language; thus, $\neg((x > 0) \neq 0)$ means $x \leq 0$. Since $\mathbf{Tree}(D_{\mathbf{LOOP}}) \in \mathbf{NonTermTrees}$, we can conclude that the program **LOOP** is non-terminating (based on Theorem 1 below). □

The soundness of predicate abstraction is stated as follows (see Appendix B for a proof).

Theorem 1. *Suppose* $\vdash P : \Gamma \rightsquigarrow D$. *If* $\mathbf{Tree}(D) \in \mathbf{NonTermTrees}$*, then* P *is non-terminating.*

4 Counterexample-Guided Abstraction Refinement (CEGAR)

This section describes our CEGAR procedure to refine abstraction based on a counterexample. Here, a *counterexample* output by a higher-order model checker is a finite subtree T of $\mathbf{Tree}(D)$, obtained by removing all but one branches of each $\texttt{br}_{\forall}$ node. Figure 5 illustrates $\mathbf{Tree}(D)$ and a corresponding counterexample (showing $\mathbf{Tree}(D) \notin \mathbf{NonTermTrees}$). In the figure, "$\cdots$" indicates an infinite path. For each $\texttt{br}_{\forall}$ node, a model checker picks one branch containing a finite path, preserving the branches of the other nodes ($\texttt{br}_{\exists}$, `call`, and `end`).

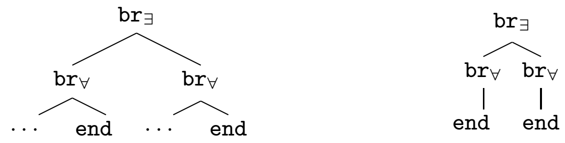


Fig. 5. $\mathbf{Tree}(D)$ (left) and a corresponding counterexample (right)

We analyze each path of the counterexample tree to infer new abstraction types for refining abstraction. To that end, we need to distinguish between two types of paths in the counterexample tree: one that has been introduced due

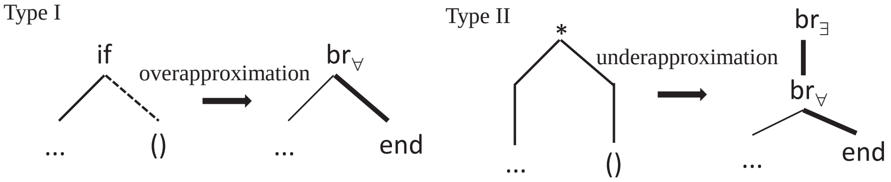


Fig. 6. Two types of paths in a counterexample

to overapproximation, and the other due to underapproximation. Figure 6 illustrates the two types. For each type, the lefthand side shows the computation tree of a source program, and the righthand side shows the tree generated by the abstract program. Thick lines show a path of a counterexample tree. In the example of Type I, the computation of a source program takes the then-branch and falls into a non-terminating computation, but predicate abstraction has introduced the spurious path taking the else branch, which was detected as a part of the counterexample. In the example of Type II, a source program generates a random number and non-deterministically branches to a non-terminating computation or a terminating computation. After predicate abstraction, the two branches by the random number generation have been merged; instead, the next deterministic computation step has been split into two by an overapproximation. This situation occurs, for example, for

$$\texttt{let}\ x : \mathbf{int}[\,] = \texttt{*}_{\texttt{int}}\ \texttt{in let}\ y : \mathbf{int}[\lambda y.y \neq 0] = x\ \texttt{in if}\ y\ \texttt{then}\ \mathit{loop}()\ \texttt{else}\ ().$$

The program generated by the abstraction is

$$\begin{aligned}\texttt{br}_\exists\{\texttt{true} \to \texttt{br}_\forall\{&\texttt{true} \to \texttt{let}\ y = \texttt{true in}\ \cdots,\\ &\texttt{true} \to \texttt{let}\ y = \texttt{false in}\ \cdots\}\}.\end{aligned}$$

Thus, the branches at $\texttt{*}_{\texttt{int}}$ in the original program have been moved to the branches at $\texttt{br}_\forall$. The classification of the paths of a counterexample into Type I or II can be performed according to the feasibility of the path, i.e., whether there is a corresponding computation path in the source program. An infeasible path is Type I, since it has been introduced by an overapproximation, and a feasible path is Type II; it has a corresponding computation path, but the two kinds of non-determinism (expressed by $\texttt{br}_\exists$ and $\texttt{br}_\forall$) have been confused by predicate abstraction. We need to refine the predicates (or, abstraction types) used for overapproximation for a Type I path, and those used for underapproximation for a Type II path. In the example program above, by refining the abstraction type for x to $\mathbf{int}[\lambda x.x \neq 0]$, we obtain

$$\begin{aligned}\texttt{br}_\exists\{&\texttt{true} \to \texttt{let}\ x = \texttt{true in br}_\forall\,\{x \to \texttt{let}\ y = \texttt{true in}\ \cdots\}\,,\\ &\texttt{true} \to \texttt{let}\ x = \texttt{false in br}_\forall\,\{\neg x \to \texttt{let}\ y = \texttt{false in}\ \cdots\}\}.\end{aligned}$$

Thus, the branches on terminating/non-terminating paths are moved to the node $\texttt{br}_\exists$.

The refinement of abstraction types based on Type I (i.e., infeasible) paths can be performed in the same way as our previous work [16]. Thus, we focus below on how to deal with a Type II path.

4.1 Dealing with Type II paths

Given a program P and a Type II path π, we first prepare fresh predicate variables $R_1, \ldots, R_k$ (called *separating predicates*), and replace each expression for random number generation $\texttt{let}\ r_i = *_{\texttt{int}}\ \texttt{in}\ e_i$ with:[4]

$$\texttt{let}\ r_i = *_{\texttt{int}}\ \texttt{in}\ \texttt{assume}(R_i(r_i)); e_i.$$

Here, an expression $\texttt{assume}(\phi); e$ evaluates to e only if ϕ is $\texttt{true}$. Then, we instantiate R_i's so that the following conditions hold.

(C1) P has no longer an execution path along π.
(C2) If the execution along π reaches $\texttt{let}\ r_i = *_{\texttt{int}}\ \texttt{in}\ \texttt{assume}(R_i(r_i)); e_i$, there is at least one value for r_i such that $R_i(r_i)$ holds.

Condition C1 is for separating the path π at $\texttt{br}_\exists$ node (recall Figure 6; the problem of a Type II path has been that terminating/non-terminating paths are merged at $\texttt{br}_\exists$ node). Condition C2 ensures that the paths separated from π are not empty. By C2, for example, an absurd assume statement like $\texttt{assume(false)}$ is excluded out. We then add the instantiations of $R_1, \ldots, R_k$ to the abstraction types for $r_1, \ldots, r_k$.

For the example

$$\texttt{let}\ x : \mathbf{int}[\,] = *_{\texttt{int}}\ \texttt{in let}\ y : \mathbf{int}[\lambda y. y \neq 0] = x\ \texttt{in if}\ y\ \texttt{then}\ \mathit{loop}()\ \texttt{else}\ ()$$

discussed above, we insert an assume statement as follows.

$$\texttt{let}\ x = *_{\texttt{int}}\ \texttt{in}\ \texttt{assume}(R(x)); \texttt{let}\ y = x\ \texttt{in if}\ y\ \texttt{then}\ \mathit{loop}()\ \texttt{else}\ ().$$

Here, the Type II path π is the one that goes through the else-branch. Thus, a condition $R(x)$ that makes it infeasible is $x \neq 0$. As a result, $\lambda x. x \neq 0$ is added to the abstraction type for x.

We sketch below how to instantiate $R_1, \ldots, R_k$. Using the technique of [16] condition (I) can be reduced to a set of non-recursive Horn clauses over predicate variables. Condition (II) is, on the other hand, reduced to constraints of the form

$$R_1(\widetilde{x}_1) \wedge \cdots \wedge R_n(\widetilde{x}_n) \wedge C \Rightarrow \exists x. R(x) \wedge C'.$$

Thus, it remains to solve (non-recursive) existentially quantified Horn clauses [4]. To solve them, we first remove existential quantification by using a Skolemization-based technique similar to [4]. We prepare a linear template of Skolem function and move existential quantifiers out of universal quantifiers. For example, given

$$\forall r. \left(\exists \nu. \nu \leq 1 \wedge R(\nu)\right) \wedge \forall r. \left(R(r) \wedge \neg(r > 0) \Rightarrow \texttt{false}\right),$$

[4] Actually, we apply the replacement to each *instance* of $\texttt{let}\ r_i = *_{\texttt{int}}\ \texttt{in}\ e_i$ along the execution path π, so that different assume conditions can be used for different instances of the same expression; we elide the details here.

program	cycle	time (msec)	program	cycle	time (msec)
`loopHO`	2	1,156	`unfoldr_nonterm`	3	13,540
`indirect_e`	1	111	`passing_cond`	2	9,202
`indirectHO_e`	1	112	`inf_clos`	2	12,264
`foldr_nonterm`	4	20,498	`fib_CPS_nonterm`	1	133
`alternate`	1	95	`fixpoint_nonterm`	2	168

Table 1. The result of the first benchmark set

we prepare the linear template $c_0 + c_1 r$ and transform the constraints into:

$$\exists c_0, c_1 . \forall \nu . r . \, (\nu = c_0 + c_1 r \Rightarrow \nu \leq 1 \land R(\nu)) \land \forall \, (R(r) \land \neg(r > 0) \Rightarrow \texttt{false}) \, .$$

We then remove predicate variables by resolution, and get:

$$\forall \nu . r . \nu = c_0 + c_1 r \Rightarrow \nu \leq 1 \land \nu > 0$$

Finally, we solve constraints in the form of $\exists \widetilde{x} . \forall \widetilde{y} . \phi$ and obtain coefficients of linear templates that we introduced in the first step. We adopt the existing constraint solving technique based [23] on Farkas' Lemma. For the running example, we obtain $c_0 = 2, c_1 = 0$ as a solution of the constraints.

Now that we have removed existential quantification, we are left with non-recursive Horn clause constraints, which can be solved by using the existing constraint solving technique [22]. For the example above, we get

$$\forall \nu . r . \, (\nu = 2 \Rightarrow \nu \leq 1 \land R(\nu)) \land (R(r) \land \neg(r > 0) \Rightarrow \bot)$$

and obtain $R = \lambda \nu . \nu > 0$ as a solution.

5 Implementation and Experiments

We have implemented a non-termination verifier for a subset of OCaml, as an extension of MoCHi [16], a software model checker for OCaml programs. We use HorSat [5] as the backend higher-order model checker, and Z3 [19] as the backend SMT solver. The web interface of our non-termination verification tool is available online [1]. We evaluated our tool by experiments on two benchmark sets: (1) test cases consisting of higher-order programs and (2) a standard benchmark set on non-termination of first-order programs [7, 18]. Both experiments were conducted on a machine with Intel Xeon E3-1225 V2 (3.20GHz, 16GB of memory) with timeout of 60 seconds. The first benchmark set and an online demo page are available from our website [1].

Table 1 shows the result of the first evaluation. The columns 'program', 'cycle', and 'time' show the name of each test case, the number of CEGAR cycles, and the elapsed time (in milliseconds), respectively. For `foldr_nonterm`, we have used a different mode for a backend constraint solver; with the default mode, our verifier has timed out. All the programs in the first benchmark set are higher-order programs; so, they cannot be directly verified by previous tools. Our tool could successfully verify all the programs to be non-terminating (except that we had to change the mode of a backend constraint solver for `foldr_nonterm`).

We explain below two of the programs in the first benchmark set: `inf_clos` and `alternate`. The program `inf_clos` is:

$$\begin{array}{l} \mathit{is\_zero}\ n = (n = 0) \qquad\qquad \mathit{succ\_app}\ f\ n = f\ (n+1) \\ f\ n\ \mathit{cond} = \mathtt{let}\ b \mathtt{=} \mathit{cond}\ n\ \mathtt{in}\ \mathtt{if}\ b\ \mathtt{then}\ (\,)\ \mathtt{else}\ f\ n\ (\mathit{succ\_app}\ \mathit{cond}) \\ \mathtt{main} = f\ \mathtt{*}_{\mathtt{int}}\ \mathit{is\_zero}. \end{array}$$

It has the following non-terminating reduction sequence:

$$\begin{array}{l} \mathtt{main} \longrightarrow^* f\,1\,\mathit{is\_zero} \longrightarrow^* f\,1\,(\mathit{succ\_app}\ \mathit{is\_zero}) \longrightarrow^* f\,1\,(\mathit{succ\_app}^2\ \mathit{is\_zero}) \\ \longrightarrow^* f\,1\,(\mathit{succ\_app}^m\ \mathit{is\_zero}) \longrightarrow^* \cdots . \end{array}$$

Note that $\mathit{succ\_app}^m\ \mathit{is\_zero}\ n$ is equivalent to $n+m = 0$; hence b in the function f always evaluates to `false` in the sequence above. For proving non-termination, we need to reason about the value of the higher-order argument *cond*, so the previous methods for non-termination of first-order programs are not applicable.

The following program `alternate` shows the strength of our underapproximation.

$$\begin{array}{l} f\ g\ h\ z = \mathtt{let}\ x \mathtt{=} \mathtt{*}_{\mathtt{int}}\ \mathtt{in}\ \mathtt{if}\ x > 0\ \mathtt{then}\ g\ (f\ h\ g)\ \mathtt{else}\ h\ (f\ h\ g) \\ \mathit{proceed}\ u = u\ (\,) \qquad \mathit{halt}\ u = (\,) \qquad \mathtt{main} = f\ \mathit{proceed}\ \mathit{halt}\ (\,) \end{array}$$

It has the following non-terminating reduction sequence:

$$\begin{array}{l} \mathtt{main} \longrightarrow^* f\ \mathit{proceed}\ \mathit{halt}\,(\,) \\ \longrightarrow^* \mathtt{if}\ \ 1 > 0\ \mathtt{then}\ \ \mathit{proceed}(f\ \mathit{halt}\ \mathit{proceed})\ \ \mathtt{else}\ \ \cdots \longrightarrow^* f\ \mathit{halt}\, \mathit{proceed}\,(\,) \\ \longrightarrow^* \mathtt{if}\ \ -1 > 0\ \mathtt{then}\ \ \cdots\ \ \mathtt{else}\ \ \mathit{proceed}(f\ \mathit{proceed}\ \mathit{halt}) \longrightarrow^* f\ \mathit{proceed}\ \mathit{halt}\,(\,) \\ \longrightarrow^* \cdots . \end{array}$$

Here, since the arguments g and h are swapped for each recursive call, the program does not terminate only if positive and negative integers are created alternately by $\mathtt{*}_{\mathtt{int}}$. Thus, the approach of Chen et al. [7] (which underapproximates a program by inserting assume statements and then uses a safety property checker to prove that the resulting program never terminates) would not be applicable. In our approach, by using the abstraction type $\mathbf{int}[\lambda x.x > 0]$ for x, f is abstracted to:

$$\begin{array}{l} f\ g\ h\ z = \mathtt{br}_\exists\{\mathtt{true} \to \mathtt{let}\ x \mathtt{=} \mathtt{true}\ \mathtt{in}\ \mathtt{br}_\forall\,\{x \to g(f\ h\ g)\}, \\ \qquad\qquad\qquad \mathtt{true} \to \mathtt{let}\ x \mathtt{=} \mathtt{false}\ \mathtt{in}\ \mathtt{br}_\forall\,\{\neg x \to h(f\ h\ g)\}\}. \end{array}$$

Thus, both branches of the if-expression are kept in the abstract program, and we can correctly conclude that the program is non-terminating.

For the second benchmark, we have borrowed a standard benchmark set consisting of 78 programs categorized as “known non-terminating examples” [7, 18]. (Actually, the original set consists of 81 programs, but 3 of them turned out to be terminating.) The original programs were written in the input language for T2 [2]; we have automatically converted them to OCaml programs. Our tool could verify 48 programs to be non-terminating in the time limit of 60 seconds. According to Larraz et al. [18], CppInv [18], T2-TACAS [7], AProVE [6, 10], Julia [21], and TNT [13] could verify 70, 51, 0, 8, and 19 programs respectively, with the same limit but under a different environment. Thus, our tool is not the best, but competitive with the state-of-the-art tools for proving non-termination of first-order programs, despite that our tool is not specialized for first-order programs. As for the comparison with T2-TACAS [7], our tool could verify 7 programs for which T2-TACAS failed, and ours failed for 10 programs that T2-TACAS could verify.

6 Related Work

Methods for disproving termination have recently been studied actively [7, 8, 13, 18]. Most of them, however, focused on programs having *finite* control-flow graphs with numerical data. For example, the state-of-the-art method by Larraz et al. [18] enumerates a strongly connected subgraph (SCSG), and checks whether there is a computation that is trapped in the SCSG using a SMT solver. Thus, it is not obvious how to extend those techniques to deal with recursion and higher-order functions. Note that unlike in safety property verification, we cannot soundly overapproximate the infinite control-flow graph of a higher-order program with a finite one.

Technically, the closest to our work seems to be the series of recent work by Cook et al. [7, 8]. They apply an underapproximation by inserting assume statements, and then either appeal to a safety property checker [7], or apply an overapproximation [8] to prove that the underapproximated program is non-terminating for all execution paths. A problem of their underapproximation [7] is that when an assume statement $\mathit{assume}(P)$ is inserted, all the computations such that $\neg P$ are discarded; so if P is wrongly chosen, they may overlook a non-terminating computation present in the branch where $\neg P$ holds. As in the case for `alternate` discussed in Section 5, in the presence of higher-order functions, there may be no proper way for inserting assume conditions. In contrast, with our predicate abstraction, given a predicate P, we basically keep both branches for P and $\neg P$, and apply an underapproximation only if the satisfiability of P or $\neg P$ is not guaranteed (recall Figure 1). In Cook et al.'s method [8], underapproximation cannot be applied after overapproximation, whereas under- and overapproximation can be arbitrarily nested in our method. Furthermore, although the framework of Cook et al. [8] is general, their concrete method can be applied to detect only non-termination in the form of lasso for programs having finite control-flow graphs. Harris et al. [14] also combine under- and overapproximation, but in a way different from ours: they use under- and overapproximation for disproving and proving termination respectively, not both for disproving termination.

There have also been studies on non-termination of term rewriting systems (TRS). Higher-order programs can be encoded into term rewriting systems, but the resulting analysis would be too imprecise. Furthermore, as mentioned in Section 1, the definition of non-termination is different.

Higher-order model checking has been recently applied to program verification [15, 16]. Predicate abstraction has been used for overapproximation for the purpose of safety property verification, but the combination of under- and overapproximation is new. Kuwahara et al. [17] have proposed a method for proving termination of higher-order programs; our new method for disproving termination plays a complementary role to that method.

The constraints generated in our CEGAR phase can be regarded as special instances of "existentially quantified Horn clauses" considered by Beyene et al. [4], where only acyclic clauses are allowed. Our constraint solving algorithm is specialized for the case of acyclic clauses. Incidentally, Beyene et al. [4] used existentially quantified clauses for verifying CTL properties of programs. Since non-termination can be expressed by the CTL formula $EG\neg terminated$, their technique can, in principle, be used for verifying non-termination. Like other methods for non-termination, however, the resulting technique seems applicable only to programs with finite control-flow graphs.

7 Conclusion

We have proposed an automated method for disproving termination of higher-order programs. The key idea was to combine under- and overapproximations by using predicate abstraction. By representing the approximation as a tree-generating higher-order program, we have reduced non-termination verification to higher-order model checking. The mixture of under- and overapproximations has also required a careful analysis of counterexamples, for determining whether and how under- or overapproximations are refined. We have implemented the proposed method and confirmed its effectiveness. Future work includes optimizations of the implementation and integration with the termination verifier [17].

Acknowledgments We would like to thank Carsten Fuhs for providing us with their experimental data and pointers to related work, and anonymous referees for useful comments. This work was partially supported by Kakenhi 23220001 and 25730035.

References

1. MoCHi(Non-termination): Model Checker for Higher-Order Programs. `http://www-kb.is.s.u-tokyo.ac.jp/~kuwahara/nonterm/`
2. T2 temporal prover. http://research.microsoft.com/en-us/projects/t2/
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI '01. pp. 203–213. ACM (2001)

4. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: CAV '13. LNCS, vol. 8044, pp. 869–882. Springer (2013)
5. Broadbent, C., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: CSL '13. LIPIcs, vol. 23, pp. 129–148 (2013)
6. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java bytecode. In: FoVeOOS '11. LNCS, vol. 7421, pp. 123–141. Springer (2012)
7. Chen, H.Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Proving nontermination via safety. In: TACAS '14. LNCS, vol. 8413, pp. 156–171. Springer (2014)
8. Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.W.: Disproving termination with overapproximation. In: FMCAD '14. pp. 67–74. IEEE (2014)
9. Emmes, F., Enger, T., Giesl, J.: Proving non-looping non-termination automatically. In: IJCAR '12. LNCS, vol. 7364, pp. 225–240. Springer (2012)
10. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with aprove. In: Proceedings of IJCAR 2014. pp. 184–191 (2014)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination in the dependency pair framework. In: Deduction and Applications. No. 05431 in Dagstuhl Seminar Proceedings (2006)
12. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV '97. LNCS, vol. 1254, pp. 72–83. Springer (1997)
13. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL '08. pp. 147–158. ACM (2008)
14. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Proceedings of SAS '10. LNCS, vol. 6337, pp. 304–319. Springer (2010)
15. Kobayashi, N.: Model checking higher-order programs. Journal of the ACM 60(3) (2013)
16. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI '11. pp. 222–233. ACM (2011)
17. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: ESOP '14. LNCS, vol. 8410, pp. 392–411. Springer (2014)
18. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using max-SMT. In: CAV '14. LNCS, vol. 8559, pp. 779–796. Springer (2014)
19. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS '08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
20. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS '06. pp. 81–90. IEEE (2006)
21. Spoto, F., Mesnard, F., Étienne Payet: A termination analyzer for java bytecode based on path-length. ACM Trans. Prog. Lang. Syst. 32(3), 8:1–8:70 (Mar 2010)
22. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP '09. pp. 277–288. ACM (2009)
23. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: POPL '13. pp. 75–86. ACM (2013)

Appendix

A The Simple Type System for the Source Language

The typing rules for the source language is given in Figure 7. In the figure, $\widetilde{\tau} \to \star$ and $\widetilde{x} : \widetilde{\tau}$ are abbreviations for $\tau_1 \to \cdots \to \tau_n \to \star$ and $x_1 : \tau_1, \ldots, x_n : \tau_n$ respectively. The rules for expressions ensure that an expression can only evaluate to the unit value.

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

$$\frac{\mathit{\Delta}(y) = \tau_1 \to \cdots \to \tau_k \to \tau \quad \mathit{\Delta} \vdash_{\mathrm{ST}} v_i : \tau_i \ (\forall i \in 1, \ldots, k)}{\mathit{\Delta} \vdash_{\mathrm{ST}} y\ v_1 \cdots v_k : \tau} \qquad \frac{\mathit{\Delta} \vdash_{\mathrm{ST}} a : \mathbf{int} \qquad \mathit{\Delta} \vdash_{\mathrm{ST}} e_1 : \star \qquad \mathit{\Delta} \vdash_{\mathrm{ST}} e_2 : \star}{\mathit{\Delta} \vdash_{\mathrm{ST}} \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \star}$$

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

$$\frac{\mathit{\Delta} = f_1 : \widetilde{\tau}_1 \to \star, \ldots, f_k : \widetilde{\tau}_k \to \star \qquad \mathit{\Delta}(\mathtt{main}) = \mathtt{unit} \quad \forall f\ \widetilde{x} = e \in P. \mathit{\Delta}, \widetilde{x}_i : \widetilde{\tau}_i \vdash_{\mathrm{ST}} e : \star \text{ (for each } i \in \{1, \ldots, k\})}{\vdash_{\mathrm{ST}} \{f_1\, \widetilde{x}_1 = e_1, \ldots, f_k\, \widetilde{x}_k = e_k\} : \mathit{\Delta}}$$

Fig. 7. Simple type system of L

B Proof of Theorem 1

This section proves Theorem 1.

We first extend the abstraction rules by the following rules:

$$\frac{\models n \neq 0 \Rightarrow \theta_{\mathit{\Gamma}} \psi_1 \qquad \models n = 0 \Rightarrow \theta_{\mathit{\Gamma}} \psi_2 \qquad \mathit{\Gamma} \vdash e_1 : \star \rightsquigarrow M_1 \qquad \mathit{\Gamma} \vdash e_1 : \star \rightsquigarrow M_2}{\mathit{\Gamma} \vdash \texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \star \rightsquigarrow \mathtt{br}_{\forall}\,\{\psi_1 \to M_1, \psi_2 \to M_2\}} \quad \text{(PA-IF-CONST)}$$

$$\frac{\models b_1 Q_1(n) \wedge \cdots \wedge b_k Q_k(n)}{\mathit{\Gamma} \vdash n : \mathbf{int}[Q_1, \ldots, Q_k] \rightsquigarrow (b_1, \ldots, b_n)} \quad \text{(PA-VALINT)}$$

These rules are required for ensuring that the abstraction relation is closed under reductions: see Lemma 1 below.

We write $\mathbf{Tree}(D, M)$ for the limit of the trees obtained from a reduction sequence $M \to_D M_1 \to_D M_2 \to_D \ldots$. By the definition, $\mathbf{Tree}(D)$ is equal to $\mathbf{Tree}(D, \mathtt{main})$.

We first prepare the following lemma.

Lemma 1 *Suppose* $\vdash P : \Gamma \rightsquigarrow D$ *and* $\Gamma \vdash e_1 : \star \rightsquigarrow M_1$. *If* $\mathbf{Tree}(D, M_1) \in \mathbf{NonTermTrees}$, *then there exist* e_2 *and* M_2 *such that* $e_1 \longrightarrow_P e_2$ *and* $\Gamma \vdash e_2 : \star \rightsquigarrow M_2$ *with* $\mathbf{Tree}(D, M_2) \in \mathbf{NonTermTrees}$.

Proof. The proof proceeds by case analysis on the shape of e_1. Since e_1 has type $\star$, there are the following cases to consider. (Note that by the assumptions, e_1 may contain only the function names defined in P as free variables.)

- Case $e_1 = (\,)$: In this case, by rule PA-Unit, M_1 must be $\mathtt{end}$. This contradicts the assumption $\mathbf{Tree}(D, M_1) \in \mathbf{NonTermTrees}$.
- Case $e_1 = f\, v_1 \cdots v_k$: In this case, we have:

  $$\begin{array}{l} M_1 = f\, V_1 \cdots V_k \\ \Gamma(f) = x_1 : \sigma_1 \to \cdots \to x_k : \sigma_k \to \sigma \\ \Gamma \vdash v_i : [v_1/x_1, \ldots, v_{i-1}/x_{i-1}]\sigma_i \rightsquigarrow V_i \\ f\, x_1 \cdots x_k = \mathtt{call}(M) \in D \\ \Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash e : \star \rightsquigarrow M \end{array}$$

  Define e_2 to be $[v_1/x_1, \ldots, v_k/x_k]e$ where $f\, x_1 \cdots x_k = e \in P$. Let M_2 be $[V_1/x_1, \ldots, V_k/x_k]M$. Then, we have $\Gamma \vdash e_2 : \star \rightsquigarrow M_2$ and $e_1 \longrightarrow_P e_2$. Furthermore, since $M_1 \longrightarrow_D \mathtt{call}(M_2)$, we have $\mathbf{Tree}(D, M_2) \in \mathbf{NonTermTrees}$ as required.
- Case $e_1 = \mathtt{let}\ x : \mathbf{int}[Q_1, \ldots, Q_k] \,\mathtt{=}\, a\ \mathtt{in}\ e_3$: In this case, we have:

  $$\begin{array}{l} M_1 = \mathtt{br}_\forall \left\{\psi_{(b_1,\ldots,b_k)} \to \mathtt{let}\ x \,\mathtt{=}\, (b_1, \ldots, b_k)\ \mathtt{in}\ M_3 \mid b_1, \ldots, b_k \in \{\mathtt{true}, \mathtt{false}\}\right\} \\ \models b_1 Q_1(a) \wedge \cdots \wedge b_k Q_k(a) \Rightarrow \psi_{(b_1,\ldots,b_k)} \\ \Gamma, x : \mathbf{int}[Q_1, \ldots, Q_k] \vdash e_3 : \star \rightsquigarrow M_3 \end{array}$$

  Here, since Γ contains no variable of integer type, a is a constant expression, and $\psi_{(b_1,\ldots,b_k)}$ is equivalent to $\mathtt{true}$ or $\mathtt{false}$. Let n be $[\![a]\!]$, and pick $(b_1, \ldots, b_k)$ such that $\models b_1 Q_1(n) \wedge \cdots \wedge b_k Q_k(n)$. Then, $\psi_{(b_1,\ldots,b_k)}$ must be equivalent to $\mathtt{true}$. Let e_2 be $[n/x]e_3$ and M_2 be $[(b_1, \ldots, b_k)/x]M_3$. Then, we have $e_1 \longrightarrow_P e_2$ and $\Gamma \vdash e_2 : \star \rightsquigarrow M_2$. Furthermore, since $M_1 \longrightarrow_D^* \mathtt{br}_\forall(\ldots, M_2, \ldots)$, we have $\mathbf{Tree}(D, M_2) \in \mathbf{NonTermTrees}$ as required.
- Case $e_1 = \mathtt{let}\ x : \mathbf{int}[Q_1, \ldots, Q_k] \,\mathtt{=}\, \mathtt{*}_{\mathtt{int}}\ \mathtt{in}\ e_3$: In this case, we have:

  $$\begin{array}{l} M_1 = \mathtt{br}_\exists \left\{\psi_{(b_1,\ldots,b_k)} \to \mathtt{let}\ x \,\mathtt{=}\, (b_1, \ldots, b_k)\ \mathtt{in}\ M_3 \mid b_1, \ldots, b_k \in \{\mathtt{true}, \mathtt{false}\}\right\} \\ \models \psi_{(b_1,\ldots,b_k)} \Rightarrow \exists x. b_1 Q_1(x) \wedge \cdots \wedge b_k Q_k(x) \\ \quad \text{(for every } b_1, \ldots, b_k \in \{\mathtt{true}, \mathtt{false}\}) \\ \Gamma, x : \mathbf{int}[Q_1, \ldots, Q_k] \vdash e_3 : \star \rightsquigarrow M_3 \end{array}$$

  By the assumption $\mathbf{Tree}(D, M_1) \in \mathbf{NonTermTrees}$, there exists $(b_1, \ldots, b_k)$ such that $\models \psi_{(b_1,\ldots,b_k)}$ and $\mathbf{Tree}(D, \mathtt{let}\ x \,\mathtt{=}\, (b_1, \ldots, b_k)\ \mathtt{in}\ M_3) \in \mathbf{NonTermTrees}$. Pick one such tuple $(b_1, \ldots, b_k)$. Then, $\exists x. b_1 Q_1(x) \wedge \cdots \wedge b_k Q_k(x)$ must hold. Pick n such that $b_1 Q_1(n) \wedge \cdots \wedge b_k Q_k(n)$. Let e_2 be $[n/x]e_3$ and M_2 be $[(b_1, \ldots, b_k)/x]M_3$. Then we have $e_1 \longrightarrow_P e_2$ and $\Gamma \vdash e_2 : \star \rightsquigarrow M_2$ with $\mathbf{Tree}(D, M_2) \in \mathbf{NonTermTrees}$ as required.

- Case $e_1 = \texttt{if}\ 0\ \texttt{then}\ e_3\ \texttt{else}\ e_4$: In this case, we have:

$$\begin{array}{ll} M_1 = \texttt{br}_\forall \{\psi_3 \to M_3, \psi_4 \to M_4\} & \\ \mathit{\Gamma} \vdash e_3 : \star \rightsquigarrow M_3 & \mathit{\Gamma} \vdash e_4 : \star \rightsquigarrow M_4 \\ \models 0 \neq 0 \Rightarrow \psi_3 & \models 0 = 0 \Rightarrow \psi_4 \end{array}$$

  By the last condition, $[\![\psi_4]\!] = \texttt{true}$. Let e_2 be e_4 and M_2 be M_4. Then, we have $e_1 \longrightarrow_D e_2$ and $\mathit{\Gamma} \vdash e_2 {:} \star \rightsquigarrow M_2$, with $\mathbf{Tree}(D, M_2) \in \mathbf{NonTermTrees}$ as required.
- Case $e_1 = \texttt{if}\ n\ \texttt{then}\ e_3\ \texttt{else}\ e_4$ with $n \neq 0$: Similar to the above case.

Theorem 1 is an immediate corollary of Lemma 1 above.

Proof of Theorem 1 Suppose $\vdash P : \mathit{\Gamma} \rightsquigarrow D$ and $\mathbf{Tree}(D) \in \mathbf{NonTermTrees}$. Then we can obtain an infinite sequence of pairs $(e_0, M_0), (e_1, M_1), \ldots$ such that $\mathit{\Gamma} \vdash e_i : \star \rightsquigarrow M_i$ and $\mathbf{Tree}(D, M_i) \in \mathbf{NonTermTrees}$ with $e_i \longrightarrow_P e_{i+1}$ as follows.

- Let $e_0 = M_0 = \texttt{main}$; and
- Given (e_i, M_i), by Lemma 1, there exist (e_{i+1}, M_{i+1}) such that $\mathit{\Gamma} \vdash e_{i+1} {:} \star \rightsquigarrow M_{i+1}$ and $\mathbf{Tree}(D, M_{i+1}) \in \mathbf{NonTermTrees}$ with $e_i \longrightarrow_P e_{i+1}$.

Thus, we have an infinite reduction sequence $\texttt{main} \longrightarrow_P e_1 \longrightarrow_P e_2 \longrightarrow_P \cdots$ as required. □