Abstract
CFL-reachability is the essence of partial correctness for recursive programs, where the qualifier CFL refers to the stack-based call/return discipline of program executions. Accordingly CFL-termination is the essence of total correctness for recursive programs. In this paper we present a program analysis method for CFL-termination. Until now, we had only program analysis methods for recursion or total correctness, but not both. We use the RHS framework [24] for interprocedural analysis to show how such methods can be integrated into a practical method for both.

Introduction
The extension of Hoare logic for reasoning about recursive programs is by now well-understood (see, e.g., [8]). In contrast, the treatment of recursion in program analysis continues to be an active research topic [3, 9–11, 13–15, 17, 23–27], as we continue to search for appropriate abstract domains for analyzing the stack as an infinite data structure. This issue is circumvented if one switches from a trace-based semantics to relational semantics (a procedure denotes a binary relation between entry and exit states). The drawback, however, is that one loses the direct connection to trace-based properties: reachability and termination, and thus partial and total correctness (or, more generally and especially for concurrent programs, safety and liveness). A breakthrough in this regard was obtained by the framework for interprocedural analysis in [24].

The contribution of the framework [24] is a refinement of the relational semantics, a refinement that accounts for traces. (Interprocedural analysis is used to compute an effective abstraction of this semantics, but this issue is orthogonal.) To be precise, the refinement of the relational semantics in [24] accounts for finite prefixes of traces, as opposed to infinite traces. As a consequence, it retrieves the connection between the relational semantics and reachability, and thus partial correctness. The framework [24] left open the question of an analogous refinement of the relational semantics that accounts for full traces and thus also retrieves the connection between the relational semantics and termination, and thus total correctness. In this paper, we do exactly that.

Related work. The technical contribution of our work is (to the best of our knowledge) the first practical interprocedural program analysis for automatic termination and total correctness proofs. Our work differs from previous work on interprocedural program analysis by the extension of its scope from partial to total correctness. Our work differs from previous work on automatic termination proofs which was restricted to non-recursive imperative programs (e.g., [4–7, 21]) or to programs in declarative languages (e.g., [18, 19]); in both those cases the above-mentioned dichotomy between the trace-based semantics and the denotational (relational) semantics is not an issue. Our work differs from existing work on model checking of temporal properties (in generalization of termination and total correctness) for finite models augmented with one stack data structure (e.g., [1, 10, 16]) by the extension of its scope to general programs.

Our TERMINATOR termination prover [7] is, in some cases, capable of proving termination of recursive programs. These are cases when a precise relationship between the interplay between the stack and states in the transitive closure of the programs transition relation are not important, as we abstract this information away in previous work. TERMINATOR can, for example, prove the termination of Ackermann’s function, while it fails to prove the termination of Fibonacci’s function.

Our work distinguishes itself from both existing interprocedural analysis and model checking by the way abstraction is introduced. It is well-known that the finitary abstraction of valuations of infinite data structures is bound to lose the termination property. Instead, one needs to abstract pairs of consecutive states (e.g., by the fact that the variable x properly decreases its value). This ‘relational abstraction’ of programs interferes in intricate ways with the abstraction of the ‘relational semantics’ of a procedure. This is one reason why it is practically mandatory to decompose the reasoning about recursion (which requires the abstraction of the ‘relational’ semantics) and the reasoning about termination (which requires ‘relational abstraction’). This decomposition does not seem possible in existing approaches, including Hoare logic proof rules for the total correctness of recursive programs (e.g., [22]). In contrast, our method achieves the decoupling of termination and recursion into two consecutive tasks. The first of the two tasks (a transformation of a recursive program into a semantically equivalent non-procedural program) is only concerned about recursion, and not termination; whereas the second is concerned only about termination, not recursion.

Partial correctness
We use the framework of [24] and follow its notation.

Programs with procedures. We assume that a program \( P \) is given by a set of procedures together with a set of global variables \( V \). We write \( g \) to refer to a valuation of global variables, and let \( G \) be the set of all such valuations. Each procedure \( p \in P \) has a set of local variables \( V_p \) that includes a program counter variable \( pc \), that ranges over the set of nodes in the procedure’s control-flow graph that we describe below. We shall omit the indexing by the procedure name when it is determined by the context and write \( pc \).

We refer to a valuation of local variables as \( l \) and write \( L_p \) for the set of all such valuations. The union over all program procedures \( \bigcup_{p \in P} L_p \) is denoted by \( L_{\bigcup P} \). Let \( g_{\text{init}} \) and \( l_{\text{init}} \) be an initial valuation of global variables and an initial valuation of local variables of some procedure, say \( p_{\text{main}} \), respectively. Without loss of generality,
A reachable computation segment is a finite consecutive sequence of variables that the corresponding program counter valuates to a start node, i.e., $l_{init}(pc) = s_{main}$.

The program nodes are labeled with program statements by a function $L$. We assume that the start node $s_p$ and the exit node $e_p$ of each procedure $p \in P$ are labeled by $\text{START}(p)$ and $\text{EXIT}(p)$, respectively. We consider three kinds of statement: operations (e.g., assignments, intra-procedural control-flow statements), procedure calls, and returns from a procedure. The corresponding labels are of the form $\text{OP}(\tau), \text{CALL}(g, \tau), \text{RETURN}(q)$, where $q \in P$ is a program procedure and $\tau$ is a transition from a set of transitions $T$. We assume that for each node $n$ labeled with $\text{CALL}(g, \tau)$ there exists a unique successor node $n'$ and that the label of $n'$ is $\text{RETURN}(q)$. For a transition $\tau$ that occurs in a node label in a procedure $p$, the corresponding binary transition relation $\rho_p$ has domain and range sets as follows:

\[
\rho_p \subseteq (G \times L_p) \times (G \times L_p), \quad \text{if } \tau \text{ occurs in } \text{OP}(\tau)
\]
\[
\rho_p \subseteq (G \times L_p) \times (G \times L_p), \quad \text{if } \tau \text{ occurs in } \text{CALL}(g, \tau).
\]

Let $\rho_{skip_p}$ be the skip transition such that $\rho_{skip_p}$ is the identity relation over the pairs of thevaluations of global and p-local variables, i.e.,

\[
\rho_{skip_p} = \{((g, l), (g, l)) \mid g \in G \land l \in L_p\}
\]

We omit the indexing if the procedure is clear from the context and $p$.

We now consider triples $(g, l, st)$, where $g \in G, l \in L_{\text{init} P}$, and $st \in L_{G, p}$. The sequence $st$ is called a stack. We write $s$ to represent the empty stack, and use $l \cdot st$ to represent a new stack with state $l$ on top of a stack $st$.

The transition relation $R$ of the program $P$ consists of pairs of such triples. We construct $R$ using transition relations that are associated with the procedure nodes in the following way:

\[
R = \{(g, l, st), (g', l', st') \mid L((pc)) = \text{CALL}(g, \tau) \land (g, l, (g, l')) \in \rho_p\}
\]
\[
\cup \{(g, l, st'), (g, l', st') \mid L((pc)) = \text{EXIT}(q)\}
\]
\[
\cup \{(g, l, st), (g', l', st) \mid L((pc)) = \text{OP}(\tau) \land (g, l, (g', l')) \in \rho_p \land (l', pc) \in E_{\text{init} P}\}
\]

By abuse of notation, the condition $((g, l), (g', l')) \in \rho_p$ here refers to the canonical extension of the relation $\rho_p$, namely from $L_p$ (the set of valuations of the local variables $l$ of a single procedure $p$) to $L_{\text{init} P}$ (the set of valuations of the local variables $l$ of all procedures). If $\tau$ occurs in the label $\text{OP}(\tau)$ of a node in procedure $p$ then the canonical extension of the relation $\rho_p$ updates only the variables in $L_p$. If $\tau$ occurs in $\text{CALL}(g, \tau)$ then it updates only the variables $L_p$.

A computation segment of the program $P$ is a consecutive sequence $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_j$ of reachable triples; i.e., $\sigma_j$ is reachable in a computation.

**Summary.** A summary is a binary relation

\[
S U M M A R Y \subseteq (G \times L_{\text{init} P}) \times (G \times L_{\text{init} P})
\]

The summary $S U M M A R Y$ contains all pairs $(g, l)$ and $(g', l')$ such that $l(pc) = s_p, l'(pc) = e_p$, and there exists a reachable computation segment $(g_1, l_1, st_1), \ldots, (g_n, l_n, st_n)$ that satisfies the following property. The segment connects $(g, l)$ with $(g', l')$, and the content of the stack at the intermediate steps is an extension of the stack content at the beginning of the segment. Formally, we require that

\[
g_1 = g, \quad l_1 = l, \quad g_n = g', \quad l_n = l', \quad st_1 = st_n,
\]
\[
st_i = st_i', st_1, \quad \text{for each } 1 < i < n, \text{ where } st_i' \in L_{\text{init} P}.
\]

Given a call node $n$, we define $S U M M A R Y(n)$ to be the projection of the summary $S U M M A R Y$ to the pairs that correspond to the procedure called at the node $n$.

Figure 1 presents the computation of summaries following [24]. For now we omit practical details, such as guaranteeing the termination of the summarization procedure via an abstraction function $\alpha$, which are standard and performed by the existing software verification tools.

The algorithm in Figure 1 computes summaries (i.e., the refined relational semantics of the program). More formally, a pair of valuations $(s_1, s_2)$ belongs to the summary $S U M M A R Y$ if and only if it is eventually added by the algorithm ("if and only if") assumes a precise abstraction $\alpha$; only one direction holds according to whether the abstraction $\alpha$ induces an over- or under-approximation. The sets $W_L, P, P'$ used in the algorithm are the usual data structures for a fixpoint-based transitive closure computation. The transitive closure is restricted to pairs of valuations $(s_1, s_2)$ whose first component has been seeded at some point. Seeding $s_1$ is encoded by adding the pair $(s_1, s_1)$ to $P'$. A valuation $s_2$ is seeded if it occurs in the second component of some pair added to the (restricted) transitive closure. (2) It is an entry valuation, i.e., its program counter value is a start node $s_p$ of some procedure $p$. [Line 3] The two conditions hold for the initial valuation $s_{init}$ given by $((g_{\text{init}}, l_{\text{init}}))$. The second condition holds by the assumption that its program counter valuates to the start node of the procedure $p_{main}$, i.e., $l_{\text{init}}(pc) = s_{main}$. The first component of every pair in the restricted transitive closure is an entry valuation; this is an invariant of the algorithm.

The summary $S U M M A R Y$ is the restriction of the transitive closure relation to pairs $(s_1, s_2)$ whose second component $s_2$ is an exit valuation, i.e., its program counter value is an exit node $e_p$ of some procedure $p$. Its first component $s_1$ is an entry valuation whose program counter value is a start node $s_p$ of the same procedure.

The transitive closure computation takes a newly added pair $(s_1, s_2)$. There are three cases according to the label of the node $s_2$.

[Line 23] In this case the node of $s_2$ is labelled with an operation. The outgoing transition $(s_2, s_3)$ is intraprocedural. The transitive closure computation is the classical one. It shortcuits two consecutive transitions by one.

In the other two cases the transitive closure computation is more complicated. This is because either the outgoing or the incoming transition follows a call edge (from a call node to a start node) in the control flow graph.

[Line 10] In this case the node of $s_2$ is a call node. The outgoing transition $(s_2, s_3)$ follows an edge from a call node to a start node (and pushes the frame of local variables of the calling procedure)}
function SUMMARIZE
input
P : program
vars
PE, PE' : path edge relations
WL : worklist
begin
1 WL := ∅
2 PE := ∅
3 PE' := α(\{(g_{init}, \text{init}), (g_{init}, \text{init})\})
4 SUMMARY := ∅
5 do
6 WL := WL \cup (PE' \setminus PE)
7 PE := PE'
8 ((g_1, l_1), (g_2, l_2)) := select and remove from WL
9 match L(l_2(pc)) with
10 | CALL(q, τ) →
11 PE' := α(PE' ∪
12 \{((g_3, l_3), (g_3, l_3)) \mid (g_2, l_2), (g_3, l_3) \in ρ_T\} ∪
13 \{((g_1, l_1), (g_4, l_2)) \mid (g_2, l_2), (g_3, l_3) \in ρ_T\}
14 exists l_4 ∈ L_{l, p} s.t.
15 (g_4, l_4), (g_1, l_1) ∈ SUMMARY\}
16 | EXIT(q) →
17 SUMMARY := SUMMARY ∪ \{((g_1, l_1), (g_2, l_2))\}
18 PE' := α(PE' ∪
19 \{((g_3, l_3), (g_2, l_4)) \mid exists l_4 ∈ L_{l, p} s.t.
20 (g_3, l_3), (g_4, l_4) \in PE'\}
21 and
22 L(l_4(pc)) = CALL(q, τ)
23 \} \in PE' \}(g_4, l_4), (g_1, l_1) \in ρ_T\}
24 | OP(τ) →
25 PE' := α(PE' ∪
26 \{((g_1, l_1), (g_3, l_3)) \mid (g_2, l_2), (g_3, l_3) \in ρ_T\}
27 \} \in PE' \}(l_2(pc), l_3(pc) \in E_{l, p}\
28 done
29 while WL ≠ ∅
return SUMMARY
end.

Figure 1. Computation of procedure summaries, following [24].
Our formulation take an abstraction function α that overapproximates binary relations over the valuations of global and local variables. The abstraction is applied on-the-fly to achieve a desired efficiency/precision trade-off.

onto the stack). Thus, s_3 satisfies the two conditions for being seeded.

[Line 14] The transitive closure computation does not shortcut two transitions (s_1, s_2) and (s_2, s_3). Instead, if the transitive closure computation has already produced a pair (s_3, s_4) whose second component is an exit valuation (thus, the pair (s_3, s_4) lies in the summary relation SUMMARY), then the transitive closure computation shortcuts the three consecutive transitions (s_1, s_2), (s_2, s_3) and (s_3, s_4). The local variables in s_2 are not changed by the procedure call.

[Line 16] In this case the node of s_2 is an exit node. Since the first component of every pair in the restricted transitive closure is an entry valuation, the entry-exit pair (s_1, s_2) belongs to the summary relation SUMMARY.

EXAMPLE 1. Consider the following program:

procedure main() begin
    l_1: z := *;
    l_2: f(z);
    l_3: exit end
procedure f(x) begin
    local y initially 0;
    l_4: if x > 0 then
        l_5: y := 2;
        l_6: while y > 0 do
            l_7: z := z - 1;
            l_8: f(x - y);
            l_9: y := y - 1;
        done
        fi
    l_10: exit end;

where x is a parameter, y is local to f, and z is a shared variable. Thus, P = \{f, main\}, V = \{z\}, V_l = \{x, y\}, V_{main} = \{\}, N_l = \{s_1, e_1, l_4, l_5, \ldots\}, N_{main} = \{s_{main}, e_{main}, l_1, \ldots\}. E_{main} = \{(s_{main}, l_1), (l_1, l_2), \ldots\}, E_l = \{(s_l, l_2), (l_4, l_5), \ldots\}. See Figure 2 for a complete picture.

The labeling map L would include, for example L(l_9) = OP(y := y - 1), where

\[ ρ_{(l_9; y := y - 1)} = \{(g, l), (g', l') \mid l'(x) = l(x) \land l'(y) = l(y) - 1\} \]

L would also include the mapping L(l_9) = CALL(f, x := -x - y).

We know, for example, that the relation

\[ \{(g, l), (g', l') \mid g(z) = l(x) - 1 \land l(y) = 2 \land g'(z) = 0 \land l'(x) = l(x) \land l'(y) = l(y)\} \]

≡ SUMMARY(l_4)

when we do not perform abstraction, i.e., α is the identity function.

Note that this example terminates, for a somewhat subtle reason: in the case we get into the loop the recursive callsite will be invoked twice, but each time with a value that is less than the current value of x, as y will equal 2 and then 1.

Equivalence wrt. Partial Correctness. We develop a procedure for replacing procedure calls at their callsites with procedure summaries:
In essence this procedure searches for callsites in the form (a) in the picture below, and replaces them with (b):

The program transformation PARTIAL.CFL replaces procedure calls by intraprocedural operations based on summaries computed by TRANSFORM:
In more detail, the label of each call node in the control flow graph is transformed into the label of the (intraprocedural) skip operation. The skip transition leads to the successor node in the control flow graph which was originally a return node but is now labeled with the (intraprocedural) operation $\text{OP} (\tau \tau')$. The transition $\tau' \tau'$ is the composition of the transition $\tau$ in the original label of the call node (the "parameter passing") and the application of the summary relation.

The resulting program $\text{PARTIALCFL}(P)$ consists of a single (non-recursive) procedure. All nodes are labeled by an intraprocedural operation $\text{OP}(\tau)$. Its transition relation does not use stack content, i.e., its computations consist of triples of the form $(g, l, \varepsilon)$ where the stack is always empty. Theorem 1 (below) provides a logical basis of such reduction. It relies on the correctness of the summary computation algorithm $\text{SUMMARIZE}$, and follows from Theorem 4.1 in [24].

**EXAMPLE 2.** Recall the simple program from Example 1, whose program graph is displayed in Figure 2. When given this graph, $\text{PARTIALCFL}$ would produce the following procedure:

```
function PartialCFL
input $P$ : program
begin
  1 SUMMARY := SUMMARIZE($P$)
  2 foreach $n \in N_{p_{main}}$ when $\mathcal{L}(n) = \text{CALL}(p, \tau)$ do
  3 TRANSFORM($p_{main}, n$)
  4 done
  5 $P \leftarrow \{p_{main}\}$
  6 return $P$
end.
```

Any partial-correctness Hoare triple $\{P\} \text{main} \{Q\}$ valid in the original program holds in the modified program.

**Theorem 1** ($\text{PARTIALCFL}$ preserves partial correctness [24]). The validity of Hoare triples for partial program correctness is preserved by the transformation $\text{PARTIALCFL}$. That is, a triple $\{\phi\} P \{\psi\}$ for the program $P$ with procedures is valid under partial correctness if and only if the triple $\{\phi\} \text{PARTIALCFL}(P) \{\psi\}$ for the program $\text{PARTIALCFL}(P)$ without procedures is valid under partial correctness, i.e.,

$$\models_{\text{par}} \{\phi\} P \{\psi\} \iff \models_{\text{par}} \{\phi\} \text{PARTIALCFL}(P) \{\psi\}.$$

The transformation $\text{PARTIALCFL}$ can be extended in a straightforward way to preserve not only Hoare triples but also the validity of assertions within procedure bodies.

**Total correctness**

See Figure 3 for a procedure, called $\text{TOTALCFL}$, that produces equivalent programs by replacing each procedure call (of the procedure $p$) by a non-deterministic choice between two intraprocedural transitions: the application of the summary and the (intraprocedural) jump transition to the start node of the procedure $p$. Informally this transformation has the effect of replacing every recursive call-site to a procedure $p$ with a conditional non-recursive command. For example, in the case of the code from Example 1, the recursive call to $f$ would be replaced in the graph-based program representation with a conditional transition which we might express in program syntax as $\text{if } * \text{ then } z := 0; \text{ else } x := x - y; \text{ goto } s_1 \text{ fi}.$

See Figure 3 for pictorial description of the transformation. For the first of the two transitions, the transformation calls the procedure $\text{TRANSFORM}$ defined previously. For the second of the two transitions, the transformation adds an edge from the call node to the entry node to the control flow graph. The original label of the entry node, $\text{START}(p)$, is replaced by the label $\text{OP}(\tau)$ with the new transition $\tau$. This transition is the union of all transitions $\tau$ in the labels $\text{CALL}(p, \tau)$ of all call nodes from which the procedure $p$ can be called (those transitions perform the "parameter passing").

The two transitions follow the two outgoing edges from what was the call node in the old program. The label of that node is updated to the operation with the skip transition. Informally, this update means that the two new transitions must accommodate the "parameter passing”.

Note that the transformation does not add an edge between the exit node and the return node to the control flow graph.

**EXAMPLE 3.** Figure 5 shows the output of $\text{TOTALCFL}$, when applied to Example 1’s program graph from Figure 2. Note that $\text{TOTALCFL}$ introduces new nodes, where commands for parameter-passing are stored. Call edges are then replaced with standard control-edges. Unlike $\text{PARTIALCFL}$, the set of reachable procedures after an application of $\text{TOTALCFL}$ remains the same—though technically the bodies are no longer treated as procedures. Note that all partial-correctness or total-correctness Hoare triples (i.e. $\{P\}\text{main}\{Q\}$ or $\{P\}\text{main}\{Q\}$) that are valid in the original program remain valid in the modified program.

**Theorem 2** ($\text{TOTALCFL}$ preserves total correctness). The validity of Hoare triples for total program correctness is preserved by the transformation $\text{TOTALCFL}$. That is, a triple $\{\phi\} P \{\psi\}$ for the program $P$ with procedures is valid under total correctness if and only if the triple $\{\phi\} \text{TOTALCFL}(P) \{\psi\}$ for the program $\text{TOTALCFL}(P)$ without procedures is valid under total correctness.
Figure 4. Program transformation $\text{TOTALCFL}$ at call, return and start nodes. (a)–(b) shows additional nodes $n'_1$ and $n'_m$ between call nodes $n_1$ and $n_m$ and a start node $s_p$. (c)–(d) demonstrates how a return node $p'$ is decorated with summaries using an operation label $\text{OP}(\tau')$. We observe that the interprocedural edges between call and start nodes are replaced by intraprocedural edges. Note that interprocedural edges between exit and return nodes are removed.

Figure 3. Program transformation $\text{TOTALCFL}$. Theorem 2 provides a logical characterization of the transformation.

correctness, i.e.,

$$\models_{\text{tot}} \{ \phi \} P \{ \psi \} \iff \models_{\text{tot}} \{ \phi \} \text{TOTALCFL}(P) \{ \psi \}.$$  

Proof (sketch) First, we prove that if the program $P$ does not terminate then there is infinite computation in the transformed program $\text{TOTALCFL}(P)$. Let $\sigma = \sigma_0, \sigma_1, \ldots$ be an infinite computation of $P$. We construct the corresponding infinite computation $\sigma'$ by traversing $\sigma$ and inspecting its triples that are labeled by call nodes using the check described below. The outcome of the check determines which branch to take when traversing the corresponding node in the program $\text{TOTALCFL}(P)$.

Let $(g_l, l, st_i)$ be a triple at the call node, i.e., $L(l) =$ CALL$(p, \tau)$. We consider the suffix of $\sigma$ that starts at $\sigma_i$. We check if it contains a triple $(g_j, l_j, st_j)$ that corresponds to the matching return node, i.e., $st_j = st_i$ and for each triple $(g_k, l_k, st_k)$ between $i$ and $j$ the stack $st_k$ is strictly larger than $st_i$. In case there is a matching triple, we follow the branch of $\text{TOTALCFL}(P)$ that corresponds to the edge $(l_p, p, l_p)$ that connects the call and return nodes in $P$. Otherwise, we follow the other branch, which goes to the start node of $p$. Following this steps we construct an infinite computation in $\text{TOTALCFL}(P)$, as we never need to follow the omitted interprocedural edges between exit and return nodes.

Now we prove that for every infinite computation of $\text{TOTALCFL}(P)$ there is a corresponding infinite computation in $P$. We apply a construction similar to the one above, but this time we traverse the infinite computation of $\text{TOTALCFL}(P)$. When visiting a triple $(g_l, l, \varepsilon)$ that corresponds to a call node with the label CALL$(p, \tau)$ in the program $P$, we look one step ahead and do the following case analysis. If the successor triple $(g_l', l', \varepsilon)$ is at the node that was present in $P$ (and hence was labeled by RETURN$(p)$), then we expand the $P$-computation $\sigma$ by adding a computation segment between $(g_l, l, \varepsilon)$ and $(g_l', l', \varepsilon)$. Such a segment exists, since the pair $(g_l, l)$ and $(g_l', l')$ appears in the summary relation that we used to construct $\text{TOTALCFL}(P)$. Otherwise, we
Figure 5. Control-flow graph from Example 3, which is the output of TOTALCFL when applied to the program in Example 1 and Figure 2. Note that SUMMARY(ℓ2) and SUMMARY(ℓ8) both represent commands with the relational meaning equaling $x' = x' \land y' = y \land z' = 0$.

proceed to the next triple and push l on the stack content in the $P$ computation.

□

EXAMPLE 4. When developing tools based on abstraction we aim to find methods in which the largest abstraction suffices in the common case. In the case of recursive functions the common case is a function with only a single recursive call (i.e., functions in which the recursive callsites do not appear in loops, and multiple recursive callsites do not occur). Consider, for example, the factorial function:

```
procedure fact(x) begin
  ℓ1: if x > 1 then
  ℓ2: y := fact(x - 1);
  ℓ3: return y * x;
  fi
  ℓ4: return 1;
end
```

This example would result in the supergraph displayed in Figure 6.

Note that even the weakest possible summary, true, suffices to prove termination in this example, as the only cyclic path through the control flow graph does not visit ℓ2r. In fact, the only case in which a summary stronger than true will be required are those in which an outer loop is used (see the example below), or the function has multiple recursive calls within it.

EXAMPLE 5. Consider a slight modification to the function from Example 1:
Figure 6. Control-flow graph of function from Example 4 after the application of TOTALCFL. Note that the choice of \textsc{SUMMARY}(\ell_2) is not important, as even the summary \texttt{true} suffices to prove termination.

\begin{verbatim}
procedure f(x) begin
  local y initially 0;
  \ell_4: if x > 0 then
  \ell_5: y := 2;
  \ell_6: while y \geq 0 do
  \ell_7: z := z - 1;
  \ell_8: f(x - y);
  \ell_9: y := y - 1;
  done
  \ell_{10}: exit;
end
\end{verbatim}

Note that, because of the change of the conditional \( y > 0 \) to \( y \geq 0 \), the procedure no longer guarantees termination. The non-terminating executions introduced by the change have the characteristic that they enter and exit infinitely often through recursive callsites. This is because all non-terminating executions visit through the recursive call after the third iteration of the loop. This case is interesting because it shows why it is crucial to consider both cases of the non-deterministic branch—the counterexample to termination in the transformed program will necessarily have to visit both sides of the non-deterministic conditional infinitely-often. To see why this is true consider the program after transformation (where we are using the sound summary \( x' = x \land y' = y \)) at the recursive callsites). See Figure 7. The only non-terminating execution in the modified program is the cycle \( s_f \rightarrow \ell_4 \rightarrow \ell_4t \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_8t \rightarrow \ell_4 \rightarrow \ell_4t \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_8t \rightarrow \ell_4 \rightarrow \ell_4t \rightarrow \ell_4 \rightarrow \ell_5 \rightarrow \ell_6 \rightarrow \ell_7 \rightarrow \ell_8 \rightarrow \ell_8t \).

\begin{verbatim}
procedure f(x) begin
  \ell_1: if x = 0 then
  \ell_2: f(1);
  \ell_3: f(0);
  \ell_{10}: return;
end
\end{verbatim}

This program’s control-flow graph, after TOTALCFL, can be found in Figure 8. If we are only considering the possibility of infinite

Figure 7. Control-flow graph fragment from code of Example 5, after application of TOTALCFL.
executions through $\ell_3$, for example, we might be tempted to drop the edge from $\ell_{2c}$ to $s_f$. The reason that this would be unsound, in this case, is that all infinite executions alternate strictly between the two callsites. Thus the program with the edge from $\ell_{2c}$ to $s_f$ removed guarantees termination.

**Example 7.** We find the summaries are also useful when proving non-termination. Consider the following example:

$$
\begin{align*}
\ell_1: & \textbf{while } x > 0 \textbf{ do} \\
\ell_2: & f(x); \\
\ell_3: & x := x + 1; \\
\end{align*}
$$

where $f$ is defined as:

$$
\begin{align*}
\textbf{procedure } f(x) \textbf{ begin} \\
\ell_4: & \textbf{ if } x > 0 \textbf{ then} \\
\ell_5: & f(x - 1); \\
\ell_6: & \textbf{return}; \\
\end{align*}
$$

This program causes termination provers such as TERMINATOR [7] to diverge, as every cyclic path is well-founded, but the program itself does not terminate. The difficulty with this program is that the number of unfoldings of $f$ tells us the value of $x$, thus every valid program path location $\ell_1$ back to $\ell_1$ contains enough information to determine the concrete values of $x$. Thus because $x' = x + 1$ and $x' \leq c$ for some concrete value $c$ determined by the length of the cycle, we know that each cycle will be well-founded, whereas the program clearly does not terminate. It is for this reason that tools such as TERMINATOR diverge while examining an infinite set of cyclic paths. Note that $x' = x$ is a sound and complete summary at $\ell_2$, thus we can use the summary to prove non-termination (using recurrence sets [12]).

**Conclusion**

We have presented a practical interprocedural program analysis for automatic termination and total correctness proofs. The primary hurdle towards this goal was the dichotomy between the trace-based semantics (for termination) and the denotational (relational) semantics (for recursion). In our method, we factor out the recursion analysis from the termination analysis. We first transform the recursive program under consideration into a semantically equivalent non-procedural program. The interprocedural reachability analysis during the first step can safely ignore the termination task; i.e., it considers only finite prefixes of traces, as opposed to (full infinite) traces as it would be required for termination. The termination analysis in the second step then uses the semantics of infinite traces of a non-procedural program.

We have implemented our new analysis method and have successfully run the implementation to automatically verify the termination of a number of interesting programs, including the ones in the paper. We have used transition-predicate abstraction [20] to implement both, the relational abstraction of the program and the abstraction of the ‘relational semantics’ of a procedure (approximating reachable computation segments for the summarization).

The immediate next question that arises from our work is how to embed the analysis method into a counterexample-guided abstraction refinement loop. This raises an interesting topic of research. We do not yet know an elegant way to pass back and forth counterexamples between the termination analysis of the nonprocedural program and the interprocedural analysis of the recursive program.

An orthogonal direction for future research is the interprocedural analysis of termination and liveness properties for concurrent programs, based on existing work for summaries for concurrent programs, e.g., [17, 23].

**References**


