Noname manuscript No.
(will be inserted by the editor)

Summarization For Termination: No Return!

Byron Cook · Andreas Podelski · Andrey Rybalchenko

Received: date / Accepted: date

Abstract We propose a program analysis method for proving termination of recursive programs. The analysis is based on a reduction of termination to two separate problems: reachability of recursive programs, and termination of non-recursive programs. Our reduction works through a program transformation that modifies the call sites and removes return edges. In the new, non-recursive program, a procedure call may non-deterministically enter the procedure body (which means that it will never return) or apply a summary statement.

Keywords Program verification · Model checking · Termination · Recursion · Summarization

1 Introduction

The extension of Hoare logic for reasoning about recursive programs is by now well-understood (see, e.g., [12]). In contrast, the treatment of recursion in program analysis continues to be an active research topic [6, 14–16, 18–20, 22, 29–33], 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 looses 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 [30].

The contribution of the framework [30] is a refinement of the relational semantics, a refinement that accounts for traces—An interprocedural analysis is used to compute

Byron Cook
Microsoft Research and Queen Mary, University of London

Andreas Podelski
University of Freiburg

Andrey Rybalchenko
Max Planck Institute for Software Systems (MPI-SWS)

an effective abstraction of this semantics, but this issue is orthogonal. To be precise, the refinement of the relational semantics in [30] 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 [30] 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., [8–11, 27]) or to programs in declarative languages (e.g., [23, 24]); 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., [2, 15, 21]) by the extension of its scope to general programs.

Our TERMINATOR termination prover [11] 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, e.g., [1, 5, 7, 13], 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 separate the reasoning about recursion (which requires the abstraction of the relational semantics) and the reasoning about termination (which requires relational abstraction).

The work in [28] extends the proof rule for termination based on the disjunctive well-foundedness of transition invariants [25], from non-recursive to recursive programs. This proof rules relies on transitive closure, which is what summaries are based on implicitly. In contrast, our work presents a program analysis method that is orthogonal to the particular termination analysis and the particular termination proof rule used in the termination analysis.

The framework of visibly pushdown languages [3, 4] can be used as an alternative formal foundation for our method. We leave an exploration of this research direction for future work.

2 Partial Correctness

We use the framework of [30] 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_p 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 $L_{\cup P}$. Let g_{init} and l_{init} be an initial valuation of global variables and an initial valuation of local variables of some procedure, say p_{main}, respectively. Without loss of generality, we assume that the procedure p_{main} is not recursive, which can be ensured by a straightforward modification of P. A procedure p is represented by a control-flow graph with a set of nodes N_p and a set of edges E_p. We assume that the sets of procedure nodes are pairwise disjoint, i.e., for each $p \neq q \in P$ we have that $N_p \cap N_q = \emptyset$. We write $E_{\cup P}$ for the set of all edges, i.e., $E_{\cup P} = \bigcup_{p \in P} E_p$. The set of nodes N_p contains a unique start node s_p and a unique exit node e_p. We assume that the return value of the procedure, if any, is passed to a global variable before the procedure exists. Since we use g_{init} and l_{init} to represent the starting point of the program, we have that the corresponding program counter valuates to a start node, i.e., $l_{\mathsf{init}}(\mathsf{pc}) = s_{p_{\mathsf{main}}}$.

The program nodes are labeled with program statements by a function $\mathcal{L}$. We assume that the start node s_p and the exit node e_p of each procedure $p \in P$ are labeled by $\mathsf{START}(p)$ and $\mathsf{EXIT}(p)$, 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 $\mathsf{OP}(\tau)$, $\mathsf{CALL}(q, \tau)$, and $\mathsf{RETURN}(q)$, where $q \in P$ is a program procedure and τ is a transition from a set of transitions $\mathcal{T}$. We assume that for each node n labeled with $\mathsf{CALL}(q, \tau)$ there exists a unique successor node n' and that the label of n' is $\mathsf{RETURN}(q)$. For a transition τ that occurs in a node label in a procedure p, the corresponding binary transition relation ρ_τ has domain and range sets as follows:

$$\begin{aligned} \rho_\tau &\subseteq (G \times L_p) \times (G \times L_p), \quad \text{if } \tau \text{ occurs in } \mathsf{OP}(\tau) \ , \\ \rho_\tau &\subseteq (G \times L_p) \times (G \times L_q), \quad \text{if } \tau \text{ occurs in } \mathsf{CALL}(q, \tau) \ . \end{aligned}$$

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

$$\rho_{\mathsf{skip}_p} = \{((g, l), (g, l)) \mid g \in G \text{ and } l \in L_p\} \ .$$

We omit the indexing if the procedure is clear from the context and write skip.

We now consider triples (g, l, st), where $g \in G$, $l \in L_{\cup P}$, and $st \in L_{\cup P}^*$. The sequence st is called a stack. We write ε 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 $\mathcal{R}$ of the program P consists of pairs of such triples. We construct $\mathcal{R}$ using transition relations that are associated with the procedure nodes in

the following way.

$$\begin{aligned}\mathcal{R} = \{((g,l,st),(g',l',l{\cdot}st)) \mid \mathcal{L}(l(\mathsf{pc})) = \mathsf{CALL}(q,\tau) \text{ and} \\ ((g,l),(g',l')) \in \rho_\tau\} \\ \cup\ \{((g,l,l'{\cdot}st),(g,l',st)) \mid \mathcal{L}(l(\mathsf{pc})) = \mathsf{EXIT}(q)\} \\ \cup\ \{((g,l,st),(g',l',st)) \mid \mathcal{L}(l(\mathsf{pc})) = \mathsf{OP}(\tau) \text{ and} \\ ((g,l),(g',l')) \in \rho_\tau \text{ and} \\ (l(\mathsf{pc}),l'(\mathsf{pc})) \in E_{\cup P}\}\end{aligned}$$

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

A computation segment of the program P is a consecutive sequence $\sigma = \sigma_0, \sigma_1, \ldots$ of triples (g,l,st). Consecutive means that each pair (σ,σ') satisfies the transition relation of the program, i.e., $(\sigma,\sigma') \in \mathcal{R}$. A computation is an initial computation segment. Initial means that the first triple is initial, i.e., $\sigma_0 = (g_{\mathsf{init}}, l_{\mathsf{init}}, \varepsilon)$. A reachable computation segment is a finite consecutive sequence $\sigma = \sigma_i, \sigma_{i+1}, \ldots, \sigma_j$ of reachable triples; i.e., σ_i is reachable in a computation.

Summarization A *summary* is a binary relation

$$\mathsf{SUMMARY} \subseteq (G{\times}L_{\cup P}){\times}(G{\times}L_{\cup P}) \ .$$

The summary $\mathsf{SUMMARY}$ contains all pairs (g,l) and (g',l') such that $l(\mathsf{pc}) = s_p$, $l'(\mathsf{pc}) = e_p$, 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

$$\begin{aligned}&g_1 = g \ , \qquad l_1 = l \ , \qquad g_n = g' \ , \qquad l_n = l' \ , \\ &st_1 = st_n \ , \\ &st_i = st_i'{\cdot}st_1 \ , \qquad \text{for each } 1 < i < n, \text{ where } st_i' \in L_{\cup P}^{+} \ .\end{aligned}$$

Given a call node n, we define $\mathsf{SUMMARY}(n)$ to be the projection of the summary $\mathsf{SUMMARY}$ to the pairs that correspond to the procedure called at the node n.

Figure 1 presents the computation of summaries following [30]. For now we omit practical details, such as guaranteeing the termination of the summarization procedure via an abstraction function α, 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 $\mathsf{SUMMARY}$ if and only if it is eventually added by the algorithm ("if and only if" assumes a precise abstraction α; only one direction holds according to whether the abstraction α induces an over- or an under-approximation). The sets WL, PE, PE' used in the algorithm are the usual data structures for a fixpoint-based transitive

```
    function SUMMARIZE
    input
     P : program
    vars
     PE, PE' : path edge relations
     WL : worklist
    begin
1    WL := ∅
2    PE := ∅
3    PE' := α({((g_init, l_init), (g_init, l_init))}
4    SUMMARY := ∅
5    do
6      WL := WL ∪ (PE' \ PE)
7      PE := PE'
8      ((g1, l1), (g2, l2)) := select and remove from WL
9      match L(l2(pc)) with
10     | CALL(q, τ) ->
11       PE' := α(PE' ∪
12                 {((g3, l3), (g3, l3)) | ((g2, l2), (g3, l3)) ∈ ρ_τ} ∪
13                 {((g1, l1), (g4, l2)) | ((g2, l2), (g3, l3)) ∈ ρ_τ and
14                                         exists l4 ∈ L_∪P such that
15                                          ((g3, l3), (g4, l4)) ∈  SUMMARY})
16     | EXIT(q) ->
17       SUMMARY := SUMMARY ∪ {((g1, l1), (g2, l2))}
18       PE' := α(PE' ∪
19                 {((g3, l3), (g2, l4)) | exists l4 ∈ L_∪P s.t.
20                                         ((g3, l3), (g4, l4)) ∈ PE' and
21                                         L(l4(pc)) = CALL(q, τ) and
22                                         ((g4, l4), (g1, l1)) ∈ ρ_τ})
23     | OP(τ) ->
24       PE' := α(PE' ∪
25                 {((g1, l1), (g3, l3)) | ((g2, l2), (g3, l3)) ∈ ρ_τ and
26                                         (l2(pc), l3(pc)) ∈ E_∪P})
27     done
28   while WL ≠ ∅
29   return SUMMARY
    end.
```

Fig. 1 Computation of procedure summaries, following [30]. 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.

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 PE'. A valuation s_1 is seeded if two conditions holds. (1) It has been recognized as reachable, which means that 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 $(g_{\mathsf{init}}, l_{\mathsf{init}})$. The second condition holds by the assumption that its program counter valuates to the start node of the

procedure p_{main}, i.e., $l_{\mathsf{init}}(\mathsf{pc}) = s_{p_{\mathsf{main}}}$. The first component of every pair in the restricted transitive closure is an entry valuation; this is an invariant of the algorithm.

The summary SUMMARY 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 of s_2.

[Line 23] In this case the node of s_2 is labeled with an operation. The outgoing transition (s_2, s_3) is intraprocedural. The transitive closure computation is the classical one. It shortcuts a 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 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.

[Line 19] As in Line 14, the transitive closure computation takes three consecutive pairs, the third one being a summary and the second one corresponding to a call edge (an edge from a call to an entry node). The only difference with Line 14 is that now the pair (s_1, s_2) is the third of the three pairs. That is, the three consecutive pairs are of the form (s_3, s_4), variables are not changed by a procedure call. This means that the local variables of s_4 are preserved by the shortcut.

Example 1 Consider the following program:

```
        procedure main() begin
ℓ1:         z := *;
ℓ2:         f(z);
ℓ3:         exit
        end

        procedure f(x) begin
            local y intially 0;
ℓ4:         if x > 0 then
ℓ5:             y := 2;
ℓ6:             while y > 0 do
ℓ7:                 z := z − 1;
ℓ8:                 f(x − y);
ℓ9:                 y := y − 1;
                done
            fi
ℓ10:        exit;
        end
```

where x is a parameter, y is local to f, and z is a shared variable. Thus, $P = \{\mathsf{f}, \mathsf{main}\}$, $V = \{\mathsf{z}\}$, $V_{\mathsf{f}} = \{\mathsf{x}, \mathsf{y}\}$, $V_{\mathsf{main}} = \{\}$, $N_{\mathsf{f}} = \{s_{\mathsf{f}}, e_{\mathsf{f}}, \ell_4, \ell_5, \ldots\}$, $N_{\mathsf{main}} = \{s_{\mathsf{main}}, e_{\mathsf{main}}, \ell_1, \ldots\}$. $E_{\mathsf{main}} = \{(s_{\mathsf{main}}, \ell_1), (\ell_1, \ell_2), \ldots\}$, $E_{\mathsf{f}} = \{(s_{\mathsf{f}}, \ell_4), (\ell_4, \ell_5), \ldots\}$. See Figure 2 for a complete picture.

The labeling map $\mathcal{L}$ would include, for example $\mathcal{L}(\ell_9) = \mathsf{OP}(\mathsf{y} := \mathsf{y} - 1)$, where

$$\rho_{(\ell_9:\mathsf{y}:=\mathsf{y}-1)} = \{((g,l),(g,l')) \mid l'(\mathsf{x}) = l(\mathsf{x}) \\ \wedge\, l'(\mathsf{y}) = l(\mathsf{y}) - 1\}$$

$\mathcal{L}$ would also include the mapping $\mathcal{L}(\ell_8) = \mathsf{CALL}(\mathsf{f}, \mathsf{x} := \mathsf{x} - \mathsf{y})$.

We know, for example, that the relation

$$\{((g,l),(g',l')) \mid g(\mathsf{z}) = l(\mathsf{x}) - 1 \\ \wedge\, l(\mathsf{y}) = 2 \\ \wedge\, g'(\mathsf{z}) = 0 \\ \wedge\, l'(\mathsf{x}) = l(\mathsf{x}) \\ \wedge\, l'(\mathsf{y}) = l(\mathsf{y})\} \\ \supseteq \\ \mathsf{SUMMARY}(\ell_8)$$

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 involked twice, but each time with a value that is less than the current value of x, as y will equal 2 and then 1.

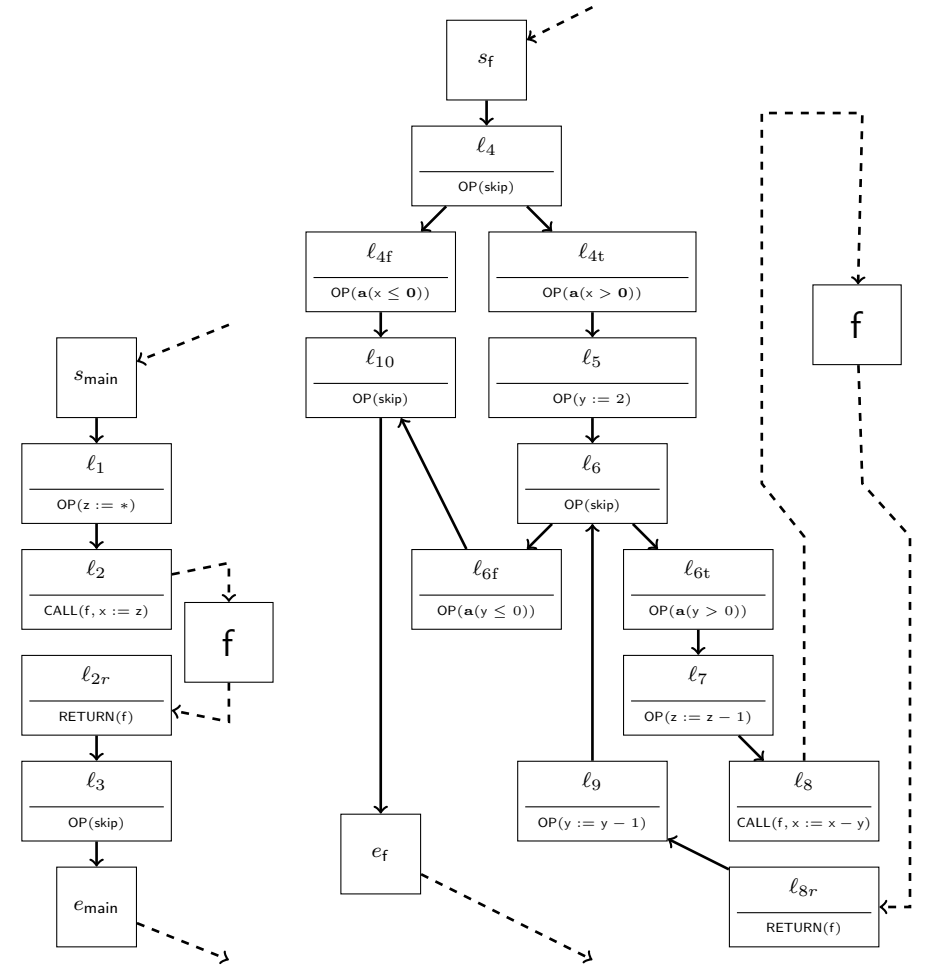
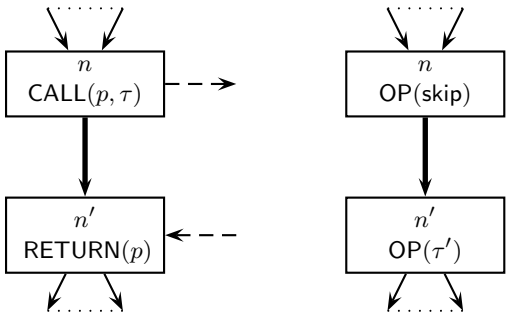


Fig. 2 Control-flow graphs for procedures main and f from Example 1. The notation $\mathbf{a}(e)$ is shorthand for **assume**(e).

Equivalence wrt. Partial Correctness We develop a procedure for replacing procedure calls at their callsites with procedure summaries:

procedure TRANSFORM
input
q : caller
n : call node
begin
1 $\mathsf{CALL}(p, \tau) := \mathcal{L}(n)$
2 $(n, n') \in E_q$
3 $\tau' :=$ fresh transition
4 $\mathcal{T} := \mathcal{T} \cup \{\tau'\}$
5 $\rho_{\tau'} := \{((g, l), (g', l)) \mid \text{exists } l' \in L_p \text{ s.t. } ((g, l), (g', l')) \in \rho_\tau \circ \mathsf{SUMMARY}\}$
6 $\mathcal{L}(n) := \mathsf{OP}(\mathsf{skip}_q)$
7 $\mathcal{L}(n') := \mathsf{OP}(\tau')$
end.

In essence this procedure searches for callsites in the form (a) in the picture below, and replaces them with (b):



(a) (b)

The program transformation PARTIALCFL replaces procedure calls by intraprocedural operations based on summaries computed by TRANSFORM:

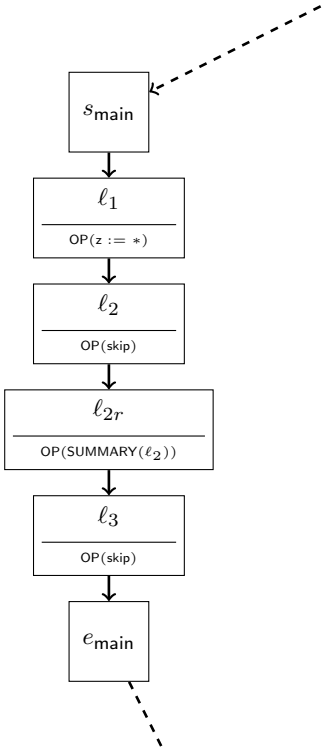
function PARTIALCFL
input
P : program
begin
1 $\mathsf{SUMMARY} :=$ SUMMARIZE(P)
2 **foreach** $n \in N_{p_{\mathsf{main}}}$ when $\mathcal{L}(n) = \mathsf{CALL}(p, \tau)$ **do**
3 TRANSFORM(p_{main}, n)
4 **done**
5 $P := \{p_{\mathsf{main}}\}$
6 **return** P
end.

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 $\mathsf{OP}(\tau')$. The transition τ' is the composition of the transition τ in the original label of the call node (the "parameter passing") and the application of the summary relation.

The resulting program PartialCFL(P) consists of a single (non-recursive) procedure. All nodes labeled by an intraprocedural operation $\mathsf{OP}(\tau)$. Its transition relation does not use stack content, i.e., its computations consist of triples of the form (g, l, ε) where the stack is always empty. Theorem 1 provides a logical basis of such reduction. It relies on the correctness of the summary computation algorithm Summarize, and follows from Theorem 4.1 in [30].

Example 2 Recall the simple program from Example 1, whose program graph is displayed in Figure 2. When given this graph, PartialCFL would produced the following procedure:



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

Theorem 1 (PartialCFL preserves partial correctness [30]) *The validity of Hoare triples for partial program correctness is preserved by the transformation* PartialCFL. *That is, a triple $\{\phi\}P\{\psi\}$ for the program P with procedures is valid under partial correctness if the triple $\{\phi\}$PartialCFL$(P)\{\psi\}$ for the program*

```
     function TotalCFL
     input
      P : program
     begin
1     SUMMARY := Summarize(P)
2     foreach p ∈ P \ {p_main} do
3       N_{p_main} := N_{p_main} ∪ N_p
4       E_{p_main} := E_{p_main} ∪ E_p
5       V_{p_main} := V_{p_main} ∪ V_p
6     done
7     foreach q ∈ P do
8       foreach n ∈ N_q when L(n) = CALL(p, τ) do
9         Transform(q, n)
10        n' := fresh node
11        N_{p_main} := N_{p_main} ∪ {n'}
12        E_{p_main} := E_{p_main} ∪ {(n, n'), (n', s_p)}
13        L(n') := OP(τ)
14      done
15    done
16    P := {p_main}
17    return P
     end.
```

Fig. 3 Program transformation TotalCFL. Theorem 2 provides a logical characterization of the transformation.

PartialCFL(P) *without procedures is valid under partial correctness, and the opposite direction holds if the abstraction function used in the summarization procedure is the identity function, i.e.,*

$$\models_{par} \{\phi\}P\{\psi\} \quad \textit{if} \quad \models_{par} \{\phi\}\textsc{PartialCFL}(P)\{\psi\} \ ,$$
$$\models_{par} \{\phi\}\textsc{PartialCFL}(P)\{\psi\} \quad \textit{if} \quad \models_{par} \{\phi\}P\{\psi\} \ \textit{and} \ \alpha(x) = \lambda x.x \ .$$

The transformation PartialCFL can be extended in a straightforward way to preserve not only Hoare triples but also the validity of assertions within procedure bodies.

3 Total correctness

See Figure 3. The program transformation TotalCFL replaces 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

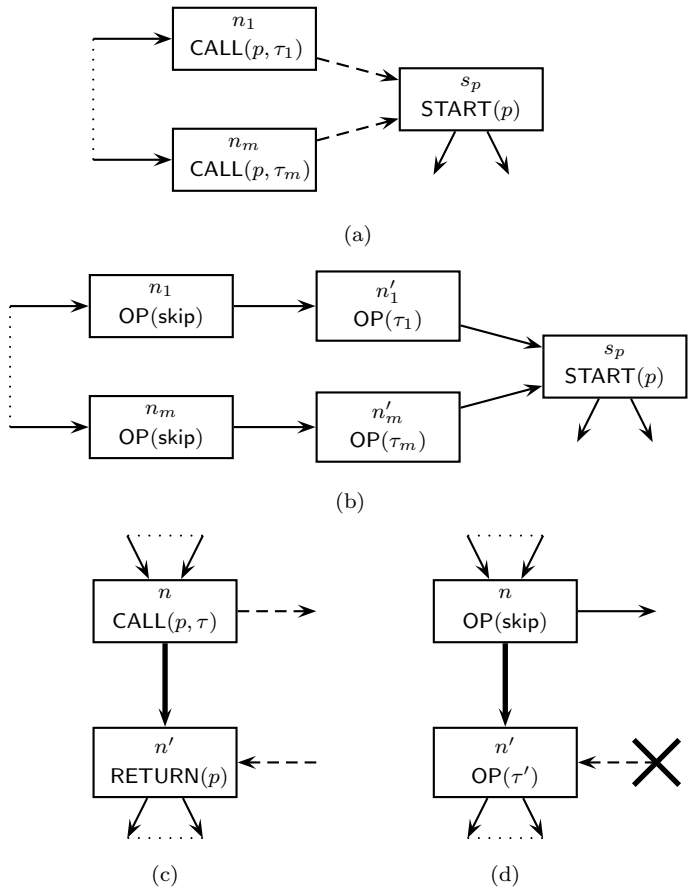


Fig. 4 Program transformation TotalCFL at call, return and start nodes. (a)–(b) shows additional nodes n'_1 and n'_m between call nodes n_1 and n'_m and a start node s_p. (c)–(d) demonstrates how a return node p' is decorated with summaries using an operation label $\mathsf{OP}(\tau')$. We observe that the interprocedural edges between call and start nodes are replaced by intraprocedural edges. Note that interprocedural edges between exit and return nodes are removed.

which we might express in program syntax as **if** $*$ **then** z $:=$ 0; **else** x $:=$ x $-$ y; **goto** s_f **fi**

See Figure 3 for the formal description of the transformation. For the first of the two transitions, the transformation calls the procedure Transform defined on Page 9. 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, $\mathsf{START}(p)$) is replaced by the label $\mathsf{OP}(\tau_s)$ with the new transition τ_s. This transition is the union of all transitions τ in the labels $\mathsf{CALL}(p, \tau)$ of all call nodes from which the procedure *sgProc* 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 TotalCFL, when applied to Example 1's program graph from Figure 2. Note that TotalCFL introduces new nodes, where commands for parameter-passing are stored. Call-edges are then replaced with standard control-edges. Unlike PartialCFL, the set of reachable procedures after an application of TotalCFL remains the same—though technically the bodies are no longer treated as procedures. Note that all partial-correctness or total-correctness Hoare triples that valid in the original program remain valid in the modified program.

Theorem 2 below holds for recursive programs over unbounded data domains. An equivalent statement for the case of finite data domains is given by Theorem 12 in [1].

Theorem 2 (TotalCFL preserves total correctness) *The validity of Hoare triples for total program correctness is preserved by the transformation* TotalCFL. *That is, a triple* $\{\phi\}P\{\psi\}$ *for the program* P *with procedures is valid under total correctness valid if and only if the triple* $\{\phi\}\text{TotalCFL}(P)\{\psi\}$ *for the program* $\text{TotalCFL}(P)$ *without procedures is valid under total correctness, , and the opposite direction holds if the abstraction function used in the summarization procedure is the identity function, i.e.,*

$$\models_{tot} \{\phi\}P\{\psi\} \quad \textit{if} \quad \models_{tot} \{\phi\}\text{TotalCFL}(P)\{\psi\} \ ,$$
$$\models_{tot} \{\phi\}\text{TotalCFL}(P)\{\psi\} \quad \textit{if} \quad \models_{tot} \{\phi\}P\{\psi\} \textit{ and } \alpha(x) = \lambda x.x \ .$$

Proof:(sketch) First, we prove that if the program P does not terminate then there is in 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 σ' by traversing σ 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_i, l_i, st_i) be a triple at the call node, i.e., $\mathcal{L}(l) = \mathsf{CALL}(p, \tau)$. We consider the suffix of σ that starts at σ_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 such a matching triple, we follow the branch of $\text{TotalCFL}(P)$ that corresponds to the edge $(l_i\mathsf{pc}, l_j\mathsf{pc})$ 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, ε) that corresponds to a call node with the label $\mathsf{CALL}(p, \tau)$ in the program P, we look one step ahead and do the following case analysis. If the

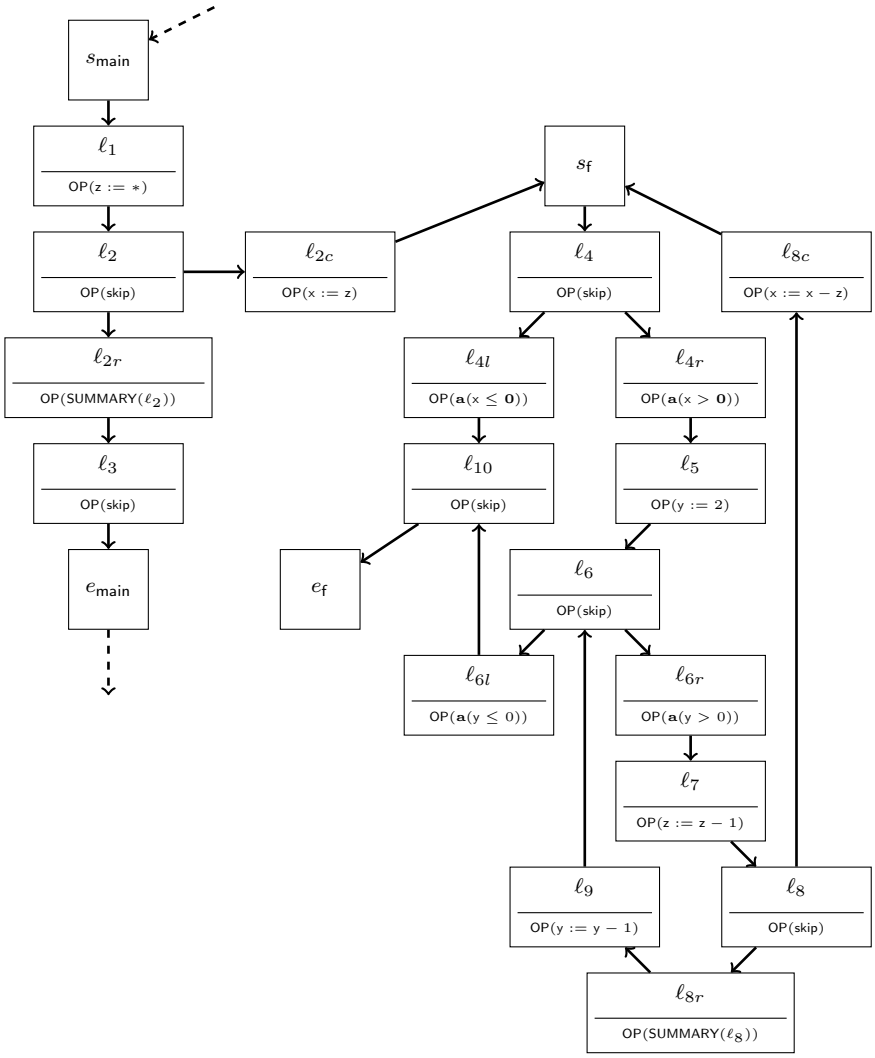


Fig. 5 Control-flow graph from Example 3, which is the output of TotalCFL when applied to the program in Example 1 and Figure 2. Note that $\mathsf{SUMMARY}(\ell_2)$ and $\mathsf{SUMMARY}(\ell_2)$ both represent commands with the relational meaning equaling $\mathsf{x}' = \mathsf{x}' \wedge \mathsf{y}' = \mathsf{y} \wedge \mathsf{z}' = 0$.

successor triple (g', l', ε) is at the node that was present in P (and hence was labeled by $\mathsf{RETURN}(p)$), then we expand the P-computation σ by adding a computation segment between (g, l, ε) and (g', l', ε). Such a segment exists, since the pair (g, l) and (g', l') appears in the summary relation that we used to construct TotalCFL(P). Otherwise, we 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

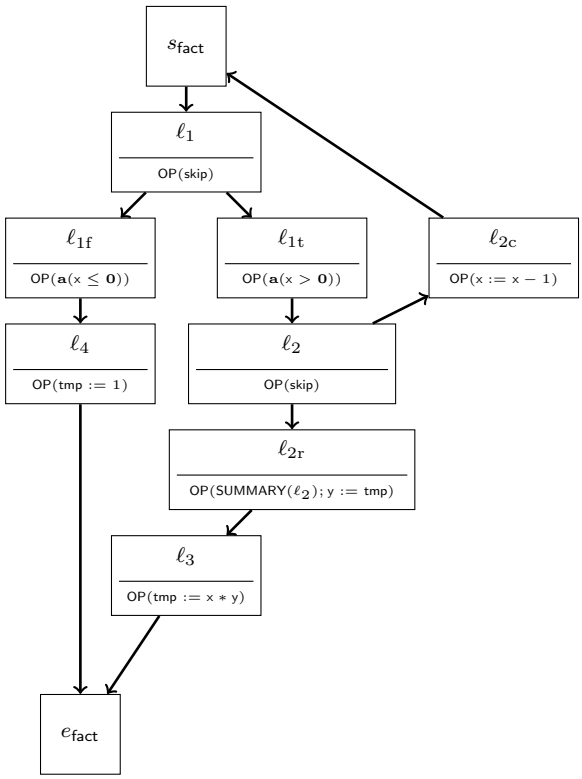


Fig. 6 Control-flow graph of function from Example 4 after the application of TotalCFL. Note that the choice of SUMMARY(ℓ_2) is not important, as even the summary **true** suffices to prove termination.

in which the recursive call sites do not appear in loops, and multiple recursive call sites 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 ℓ_{2r}. 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:

```
                  procedure f(x) begin
                      local y intially 0;
ℓ4:                   if x > 0 then
ℓ5:                       y := 2;
ℓ6:                       while y ≥ 0 do
ℓ7:                           z := z − 1;
ℓ8:                           f(x − y);
ℓ9:                           y := y − 1;
                          done
                      fi
ℓ10:                  exit;
                  end
```

Note that, because of the change of the conditional $\mathsf{y} > 0$ to $\mathsf{y} \geq 0$, 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 of recursive call sites. 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 its 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 $\mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}$ at the recursive call sites. See Figure 7. The only non-terminating execution in the modified program is the cycle $s_{\mathsf{f}} \to \ell_4 \to \ell_{4\mathrm{t}} \to \ell_5 \to \ell_6 \to \ell_7 \to \ell_8 \to \ell_{8\mathrm{r}} \to \ell_4 \to \ell_{4\mathrm{t}} \to \ell_6 \to \ell_7 \to \ell_8 \to \ell_{8\mathrm{r}} \to \ell_4 \to \ell_{4\mathrm{t}} \to \ell_6 \to \ell_7 \to \ell_8 \to \ell_{8\mathrm{c}}$

Example 6 When proving a liveness property such as "call site location ℓ_3 can only be visited infinitely-often", the reader may be tempted to modify the transformation at other call sites such that only the summary is used, and not the added edge back to the beginning of the procedure. Such an optimization would be unsound, as this example shows:

```
                  procedure f(x) begin
ℓ1:                   if x = 0 then
ℓ2:                       f(1);
                      else
ℓ3:                       f(0);
                      fi
ℓ4:                   return;
                  end
```

This program's control-flow graph, after TotalCFL, can be found in Figure 8. If we are only considering the possibility of infinite executions through ℓ_3, for example, we might

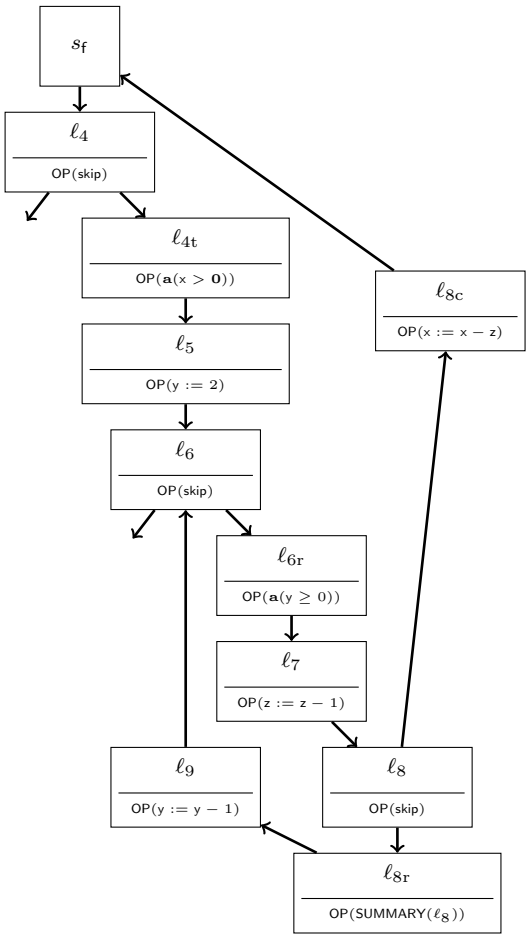


Fig. 7 Control-flow graph fragment from code of Example 5, after application of TotalCFL.

be tempted to drop the edge from ℓ_{2c} to s_{f}. The reason that this would be unsound, in this case, is that all infinite executions alternates strictly between the two call sites. Thus the program with the edge from ℓ_{2c} to s_{f} removed guarantees termination.

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

ℓ_1: **while** x > 0 **do**
ℓ_2: f(x);
ℓ_3: x := x + 1;
od

where f is defined as:

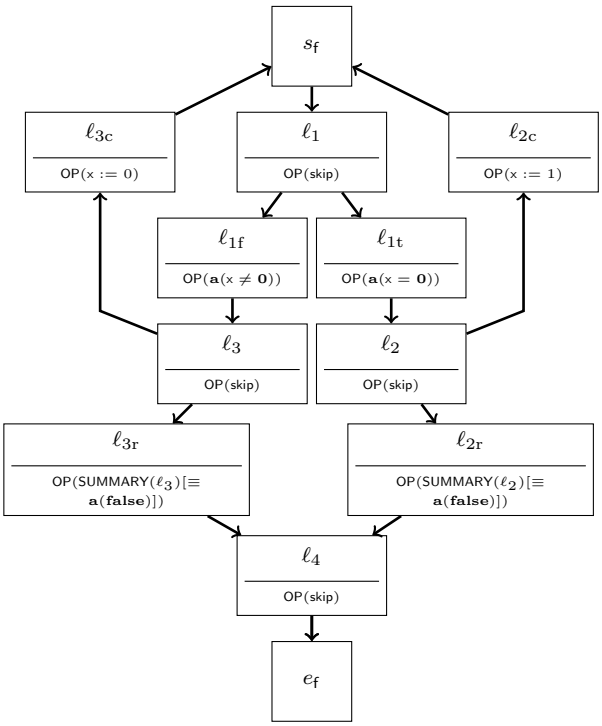


Fig. 8 Control-flow graph of function from Example 6 after the application of TotalCFL.

```
        procedure f(x) begin
ℓ4:         if x > 0 then
ℓ5:             f(x − 1);
            fi
ℓ6:         return;
        end
```

This program causes termination provers such as Terminator [11] 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 ℓ_1 back to ℓ_1 contains enough information to determine the concrete values of x. Thus because $\mathsf{x}' = \mathsf{x} + 1$ 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 $\mathsf{x}' = \mathsf{x}$ is a sound *and* complete summary at ℓ_2, thus we can use the summary to prove non-termination (using recurrence sets [17]).

4 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.

Our implementation uses transition-predicate abstraction [26] to implement both the relational abstraction of the program and the abstraction of the relational semantics of procedures (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 non-procedural 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., [22, 29].

References

1. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 2005.
2. R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *POPL*. ACM, 2006.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*. Springer, 2004.
4. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*. ACM, 2004.
5. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN*. Springer, 2000.
6. T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, 2001.
7. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*. Springer, 1997.
8. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*. ACM, 1993.
9. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
10. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, 2002.
11. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*. ACM, 2006.
12. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1989.

13. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*. Springer, 2000.
14. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL*. ACM, 2000.
15. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV*. Springer, 2001.
16. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*. Springer, 2006.
17. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*. ACM, 2008.
18. B. Jeannet, A. Loginov, T. W. Reps, and S. Sagiv. A relational approach to interprocedural shape analysis. In *SAS*. Springer, 2004.
19. R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*. ACM, 2007.
20. J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*. ACM, 2004.
21. A. Lal and T. W. Reps. Improving pushdown system model checking. In *CAV*. ACM, 2006.
22. A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*. Springer, 2008.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.
24. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *CAV*. Springer, 2006.
25. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*. IEEE, 2004.
26. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, 2005.
27. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
28. A. Podelski, I. Schaefer, and S. Wagner. Summaries for total correctness of recursive programs. In *ESOP*. Springer, 2005.
29. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*. ACM, 2004.
30. T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
31. T. W. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*. Springer, 2007.
32. T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 2005.
33. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Application*. Prentice-Hall International, 1981.