Summaries for While Programs with Recursion

Andreas Podelski   Ina Schaefer   Silke Wagner

MPI–I–2004–1–007   December 2004

Authors' Addresses

Andreas Podelski
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
`podelski@mpi-sb.mpg.de`

Ina Schaefer
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
`ina.schaefer@mpi-sb.mpg.de`

Silke Wagner
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
`silke.wagner@mpi-sb.mpg.de`

Abstract

Procedure summaries are an approximation of the effect of a procedure call. They have been used to prove partial correctness and safety properties. In this paper, we introduce a generalized notion of procedure summaries and present a framework to verify total correctness and liveness properties of a general class of while programs with recursion. We provide a fixpoint system for computing summaries, and a proof rule for total correctness of a program given a summary. With suitable abstraction methods and algorithms for efficient summary computation, the results presented here can be used for the automatic verification of termination and liveness properties for while programs with recursion.

Keywords

Recursive Programs, Summaries, Total Correctness, Program Analysis, Program Verification

1 Introduction

Procedure summaries are a fundamental notion in the analysis and verification of recursive programs [20, 18, 3]. They refer to the approximation of the "functional" effect of a procedure call. So far, they have shown useful for deriving and proving partial correctness, invariance and safety properties ("nothing bad may happen"). The results in this paper show that procedure summaries may be useful for deriving and proving termination and liveness properties ("something good will happen").

More specifically, we present a notion of summaries that applies to general programs with arbitrary nesting of while loops and recursion; the program variables range over possibly infinite data domains. A summary captures the effect of the unbounded unwinding of the body of procedure definitions, as well as of while loops. More generally, a summary may refer to any pair of programs points and captures the effect of computations that start and end at these program points.

We may use a pair of state assertions to express a summary, e.g. the pair $(x > 0, x < 0)$ to describe that the program variable x is first positive and then negative. We also may use assertions on state pairs, e.g. the assertion $x' = -x$ to describe that the program variable x gets multiplied by -1.

It is obvious that partial correctness and invariance and safety properties can be expressed in terms of summaries. This paper shows that also termination can be expressed in terms of summaries. We here concentrate on termination; the reduction of more general liveness properties to termination would follow the lines of [22, 14, 15].

The two classical proof rules for partial correctness and termination use invariants and variants (ranking functions) for the auxiliary assertion on the program. We present a proof rule for total correctness that uses summaries for the (one) auxiliary assertion on the program. Besides illustrating a new facet of total correctness of recursive programs, the contribution of the proof rule lies in its potential for automation via abstract interpretation [8, 9]. The considerable investment of research into the efficient computation of summaries has been a success; its payoff through industrialized tools checking invariance and safety properties of recursive programs [3] may well extend to termination and liveness properties. We believe that our paper may lead to several directions of follow-up work towards that goal.

2 Related Work

Among the vast amount of work on the analysis and verification of recursive programs, we will cover the part that seems most relevant for ours. In short, to advance a sum-up of the comparison, none of that work considers a notion of summary as general as ours (which refers to arbitrarily precise descriptions of the effect of computations between general pairs of program points of general while programs), and none of that work exploits summaries for termination.

Hierarchical State Machines (HSMs) [5], called Recursive State Machines (RSMs) in [2], are a model of recursive programs over finite data domains (and hence with finitely many *states*, if state refers to the valuation s of the program variables, i.e. without the stack contents γ; in our technical exposition, we use *configuration* to refer to the pair (s, γ) and avoid the term 'state' altogether).

As a side remark, we note that while loops are irrelevant in finite-state programs such as HSMs or RSMs, and can be eliminated in programs with recursion. Our exposition (for programs with while loops and recursion) permits to compare summaries for while loops with the summaries for recursive procedures replacing them.

The model checking algorithms in [5] and in [2] account for temporal properties including termination and liveness. Hence, one may wonder whether one can not prove those properties for general recursive programs by first abstracting them to finite-state recursive programs (using e.g. predicate abstraction as in [3]) and then applying those model checking algorithms. The answer is: no, one can not. Except for trivial cases, the termination or liveness property gets lost in the abstraction step. In the automation of our proof rule by abstract interpretation, one may use the idea of transition predicate abstraction [15] to obtain abstractions of summaries; a related idea, developed independently, appears in [11].

The model checking algorithms in [5] and in [2] are based on the automata-theoretic approach. In [5], the construction of a monitor Buechi automaton for the LTL or CTL* property is followed by a reachability analysis for the monitored HSM in two phases. First, summary edges from call to return of a module and path edges from entry nodes of a module to an arbitrary node in the same module are constructed. Additionally, it is indicated whether those paths pass an accepting state of the monitor. Second, the graph of a Kripke structure augmented with summary and path edges is checked for cycles. If a cycle through an accepting path exists the Buechi acceptance condition is satisfied and the property fails.

In [5], the construction of summary edges follows the fundamental graph-theoretic set-up of [18]. In [2], a (closely related) setup of Datalog rules is used. The fixpoint system that we use (in our proof rule in order to validate a summary for a given program) are reminiscent of those Datalog rules; for a rough comparison one may say that we generalize the Datalog rules from propositional to first-order logic. This is needed for the incorporation of infinite data types, which in fact is mentioned as a problem for future work in [2].

The CaRet logic in [1] expresses properties of recursive state machines, such as non-regular properties concerning the call stack, that go beyond the properties considered in this paper (which refer to program variables only). The model checking algorithm for CaRet presented in [1] uses summary edges for procedures as in [2] and is again restricted to finite data types.

The model checker Bebob [4], a part of the SLAM model checking tool [3], is based on the construction of procedure summaries adapted from [18] using CFL-reachability. The applied algorithm is again a two stage process. First, path and summary edges are constructed and then, the actual reachability analysis is carried out by using summary and path edges. Bebop applies to C-like structured programs with procedures and recursion and no other than Boolean variables.

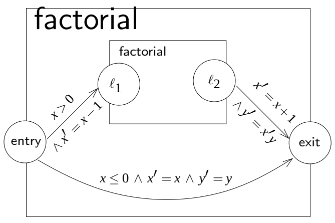
The work presented here is related to the work on program termination in [13, 14, 15] in the following way. The notion of transition invariants introduced in [14] for characterizing termination can be instantiated for recursive programs in either of two ways, by referring to program valuations (i.e. without stack contents) or by referring to configurations (i.e. pairs of program valuations and stack contents). Either case does not lead to useful proof rules for total correctness. The notion of summaries, and its putting to use for termination proofs for recursive programs, are contributions proper to this paper. The work in [14] and in [15] is relevant for the automation of our proof rule in two different ways. The algorithm presented in [13] can be used to efficiently check the third condition of the proof rule. As mentioned above, the abstraction investigated in [15] can be used to approximate summaries (and thus automate their construction by least-fixpoint iteration).

As pointed out by an anonymous referee, it is possible to define summaries using the formalism of so-called weighted pushdown systems [6, 19]. This would be useful in order to give an alternative view on our results in this framework.

3 Examples

We consider the program factorial below. We will construct a summary for the program and use the summary for proving total correctness. We hereby informally instantiate the proof rule that we will introduce in Section 6. The semantics of procedure calls is call by reference.

```
factorial(x,y)  =
entry :           if   x > 0
                          {
                           x = x − 1;
                  ℓ1 :     factorial(x,y);
                  ℓ2 :     x = x + 1;
                           y = x · y;
                          }
exit :
```


factorial
factorial
ℓ_1
ℓ_2
entry
exit
$x > 0 \wedge x' = x - 1$
$x' = x + 1 \wedge y' = x'y$
$x \leq 0 \wedge x' = x \wedge y' = y$

In the abstract notation used in this paper, the program consists of one module M_0 given by a set Cmds_0 of three *commands* and a set Calls_0 of one *call*.

$$
\begin{array}{rlccl}
\mathsf{Cmds}_0 = \{ & (\mathsf{entry}, & x \leq 0 \wedge x' = x \wedge y' = y\,, & \mathsf{exit})\,, & \\
 & (\mathsf{entry}, & x > 0 \wedge x' = x-1 \wedge y' = y\,, & \ell_1)\,, & \\
 & (\ell_2, & x' = x+1 \wedge y' = x'y\,, & \mathsf{exit}) & \} \\
\mathsf{Calls}_0 = \{ & (\ell_1, & 0, & \ell_2) & \}
\end{array}
$$

The one-step transition relation R over program valuations is specified by the assertions $R1$ to $R5$ below. The assertions $R1$ to $R3$ correspond to the execution of the commands in $Cmds_0$ (and are obtained by their direct translation). The assertions $R4$ and $R5$ correspond to the execution of a call; we will see further below how we can obtain $R4$ and $R5$.

As usual, we express a binary relation over program valuations as a set of valuations of the program variables and the primed version of the program variables. The program variables include the program counter pc which ranges over the four locations (entry, exit, ℓ_1 and ℓ_2) of the program.

$$
\begin{array}{ll}
R1 & \mathsf{pc} = \mathsf{entry} \wedge x \leq 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \mathsf{exit} \\
R2 & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x - 1 \wedge \mathsf{pc}' = \ell_1 \\
R3 & \mathsf{pc} = \ell_2 \wedge x' = x+1 \wedge y' = x'y \wedge \mathsf{pc}' = \mathsf{exit} \\
R4 & \mathsf{pc} = \ell_1 \wedge x \leq 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \ell_2 \\
R5 & \mathsf{pc} = \ell_1 \wedge x > 0 \wedge x' = x \wedge y' = (x-1)!xy \wedge \mathsf{pc}' = \ell_2
\end{array}
$$

We next consider execution sequences that contain no or *finished* recursive calls (where the final stack of the execution sequence is again the same as the initial one). The corresponding transition relation T is specified by the assertions below in addition to the relations $R1 - R5$. The assertions $T1$ and $T2$ apply to pairs of program valuations at entry and exit. The assertions $R4$ and $R5$ apply to pairs of program valuations at ℓ_1 and ℓ_2. We obtain $R4$ and $R5$ by replacing in $T1$ and $T2$ the conjuncts $\mathsf{pc} = \mathit{entry}$ and $\mathsf{pc}' = \mathsf{exit}$ by the conjuncts $\mathsf{pc} = \ell_1$ and $\mathsf{pc}' = \ell_2$.

$$
\begin{array}{ll}
T1 & \mathsf{pc} = \mathsf{entry} \wedge x \leq 0 \wedge x' = x \wedge y' = y \wedge \mathsf{pc}' = \mathsf{exit} \\
T2 & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x \wedge y' = (x-1)!xy \wedge \mathsf{pc}' = \mathsf{exit} \\
T3 & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x - 1 \leq 0 \wedge \mathsf{pc}' = \ell_2 \\
T4 & \mathsf{pc} = \mathsf{entry} \wedge x > 0 \wedge x' = x - 1 > 0 \wedge y' = (x-2)!\,(x-1)y \wedge \mathsf{pc}' = \ell_2 \\
T5 & \mathsf{pc} = \ell_1 \wedge x \leq 0 \wedge x' = x+1 \wedge y' = (x+1)y \wedge \mathsf{pc}' = \mathsf{exit} \\
T5 & \mathsf{pc} = \ell_1 \wedge x > 0 \wedge x' = x+1 \wedge y' = x!\,(x+1)y \wedge \mathsf{pc}' = \mathsf{exit}
\end{array}
$$

Finally, we consider multiple-step execution sequences with *unfinished* recursive calls (i.e. where the final stack of the execution sequence has increased by at least one item). The corresponding transition relation S is specified by assertions such as $S1$ and $S2$ below (we omit the other S-assertions).

$$\begin{array}{ll} S1 & \mathsf{pc} = \mathsf{entry} \wedge x \geq 0 \wedge x' < x \\ S2 & \mathsf{pc} = \ell_1 \wedge x \geq 0 \wedge x' < x \end{array}$$

The disjunction of R-, S- and T-assertions is a *summary* of the factorial program. The total correctness, specified by the pair of the precondition and the postcondition

$$\begin{array}{rcl} \mathsf{pre} & \equiv & \mathsf{pc} = \mathsf{entry} \wedge x \geq 0 \wedge y = 1 \\ \mathsf{post} & \equiv & \mathsf{pc}' = \mathsf{exit} \wedge y' = x! \end{array}$$

follows, by the proof rule presented in Section 6, from two kinds of basic observation on the summary.

(1) The assertion $T1 \vee T2$ in conjunction with the assertion pre entails the assertion post. (2) Each assertion denotes a well-founded relation. This is true for the assertion $S1$ by a classical argument, and it is trivially true for each of the other assertions presented here (since a relation with pairs of different locations ℓ and ℓ' admits only chains of length 1).

Second Example: Insertion Sort. In this example, reasoning over termination must account for the nesting of recursive calls and while loops. Given an array A and a positive integer n the ins_sort program sorts A. The procedure insert is applied to an array of size n and uses a while loop to insert its nth element $A[n-1]$ in its proper place, assuming that the first $n-1$ elements are sorted.

```
ins_sort(A,n)  =                          insert(A,n)  =
  entry_0 :    if n ≤ 1 then A              entry_1 :    i = n;
               else                         ℓ_4 :        while (n > 1  &
                 {                                              A[n−1] < A[n−2])
                   n = n − 1;                                 {
  ℓ_1 :            ins_sort(A,n);                               swap(A[n−2],A[n−1]);
  ℓ_2 :            n = n + 1;                                   n = n − 1;
  ℓ_3 :            insert(A,n);                               }
                 }                          ℓ_5 :          n = i;
  exit_0 :                                  exit_1 :
```

A summary of the ins_sort program must account for execution sequences with nested recursion and unfolding of while loops. Again, we give a summary for the program in the form of a disjunction of R-, S- and T-assertions; see below for the ones that are most interesting for the total correctness proof.

$T1$	$\mathsf{pc} = \mathsf{entry}_0 \wedge n \leq 1 \wedge \mathsf{pc}' = \mathsf{exit}_0$
$T2$	$\mathsf{pc} = \mathsf{entry}_0 \wedge A'[0] \leq A'[1] \leq \ldots \leq A'[n-1] \wedge \mathsf{pc}' = \mathsf{exit}_0$
$T3$	$\mathsf{pc} = \ell_4 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \ell_4$
$S1$	$\mathsf{pc} = \mathsf{entry}_0 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \mathsf{entry}_0$
$S2$	$\mathsf{pc} = \ell_1 \wedge n > 0 \wedge n' < n \wedge \mathsf{pc}' = \ell_1$

Total correctness follows from the same two kinds of properties of the summary as in the previous example. The assertions $T1$ and $T2$ imply partial correctness if n is equal to the length of the array. Termination follows from the well-foundedness of $T3$ (which accounts for computation sequences in the while loop) and $S1$ and $S2$ (which account for the recursive descend). Note that the well-foundedness argument is itself detached from the account for (possibly nested) recursion and loops; it is applied to each assertion in isolation.

4 Recursive Programs

In this section we fix the abstract notation for general while programs with recursion. It should be straightforward to map the concrete syntax of an imperative programming language into this notation. In the remainder of the paper, we assume to have an arbitrary but fixed program $\mathcal{P}$.

- The program consists of a set of *modules* $\{M_0, \ldots, M_m\}$.
- The set of *locations* of the module M_j is denoted by Loc_j.
- Each module $\mathcal{M}_j$ has two distinguished locations noted entry_j and exit_j which are its unique *entry point* and its unique *exit point*.
- Each *command* of a module is a triple (ℓ_1, c, ℓ_2) consisting of the locations ℓ_1 and ℓ_2 of the module (the *before* and the *after* location) and the transition constraint c. A transition constraint is a formula over primed and unprimed program variables.
- Each *call* of a module is a triple (ℓ_1, k, ℓ_2) consisting of the locations ℓ_1 and ℓ_2 of the module (the *call* location and the *return* location) and the index k of the module being called (i.e. $k \in \{0, \ldots, m\}$).

The sets Cmds and Calls consist of the commands and calls, respectively, of all modules of the program. The set Loc consists of its locations, i.e. $\mathsf{Loc} = \mathsf{Loc}_0 \cup \ldots \cup \mathsf{Loc}_m$.

The set Var consists of the program variables, which usually range over unbounded data domains. The set Var' contains the primed versions of the program variables. We use an auxiliary variable, the program counter pc, which ranges over the finite set Loc of locations of all modules.

A *program valuation* ("state") s is a valuation for the program variables and the program counter, i.e. s is a mapping from $\mathsf{Var} \cup \{\mathsf{pc}\}$ into the union of data domains. We note Σ the set of all program valuations.

A *configuration* $q = (s, \gamma)$ is a pair of a program valuation s and a word γ (the stack) over the alphabet Loc of program locations of all modules. We note Q the set of configurations; formally, $Q = \Sigma \times \mathsf{Loc}^\star$.

In assertions we use γ as a "stack variable", i.e. a variable that ranges over $\mathsf{Loc}^\star$. An assertion (e.g. a first-order formula) over the set of variables $\mathsf{Var} \cup \{\mathsf{pc}\} \cup \{\gamma\}$ denotes a set of configurations. For example, the set of initial configurations is denoted by the assertion $\mathsf{pc} = \mathsf{entry}_0 \wedge \gamma = \varepsilon$ where entry_0 is the entry location of the designated 'main' module M_0 and ε is the empty stack. An assertion over the set of variables $\mathsf{Var} \cup \{\mathsf{pc}\} \cup \{\gamma\} \cup \mathsf{Var}' \cup \{\mathsf{pc}'\} \cup \{\gamma'\}$ denotes a binary relation over configurations.

We note $\rightsquigarrow$ the *transition relation over configurations*, i.e. $\rightsquigarrow \subseteq Q \times Q$. The three different types of transitions are: local transition inside a single module, call of another module and return from a module. The transition relation $\rightsquigarrow$ is denoted by the disjunction of the assertions below.

$$\begin{array}{lllll}
\mathsf{pc} = \ell_1 & \wedge\ \mathsf{pc}' = \ell_2 & \wedge \quad c & \wedge\ \gamma' = \gamma & \text{where } (\ell_1, c, \ell_2) \in \mathsf{Cmds} \\
\mathsf{pc} = \ell_1 & \wedge\ \mathsf{pc}' = \mathsf{entry}_j & \wedge\ \mathsf{Var}' = \mathsf{Var} & \wedge\ \gamma' = \ell_2.\gamma & \text{where } (\ell_1, j, \ell_2) \in \mathsf{Calls} \\
\mathsf{pc} = \mathsf{exit}_j & \wedge\ \mathsf{pc}' = \ell_2 & \wedge\ \mathsf{Var}' = \mathsf{Var} & \wedge\ \gamma = \ell_2.\gamma' & \text{where } (\ell_1, j, \ell_2) \in \mathsf{Calls}
\end{array}$$

According to the three kinds of assertions, we distinguish three kinds of transitions.

A *local transition* $q \rightsquigarrow q'$ is induced by a command (ℓ_1, c, ℓ_2) of the module. It is enabled in the configuration q if the values of the program variables satisfy the guard formula in the transition constraint c of the command at the corresponding location ℓ_1. The program counter and the program variables are updated in q' accordingly; the stack remains unchanged.

Both, a *call* and a *return transition* $q \rightsquigarrow q'$, are induced by a call command (ℓ_1, j, ℓ_2) calling a module M_j. In both, the stack γ is updated and the program variables remain unchanged ($\mathsf{Var}' = \mathsf{Var}$ stands for the conjunction of $x' = x$ over all program variables x).

In a *call* transition the stack is increased by the return location ℓ_2 (by a *push* operation). The value of the program counter is updated to the entry location entry_j of the module M_j being called.

When the exit location of the called module M_j is reached, the control flow returns to the return location ℓ_2 of the calling module, which is the top value of the return stack. Thus, in a *return* transition, the value of the program counter is updated by the top value of the stack, and the stack is updated by removing its top element (by a *pop* operation).

A (possibly infinite) *computation* is a sequence of configurations $q_0, q_1, q_2, \ldots$ that starts with an initial configuration and that is consecutive, i.e. $q_i \leadsto q_{i+1}$ for all $i \geq 0$.

5 Summaries

In its generalized form that we introduce in this section, a summary captures the effect of computations that start and end at any pair of program points (and not just to the pair of the entry and exit points of a module). The computations in questions may contain calls that are not yet returned; i.e., in general they don't obey to the '*each call is matched by a subsequent return*' discipline. We first introduce the corresponding *transition relation over program valuations* the *descends* relation, noted $\xrightarrow{\leq}$.

Definition 1 (Intraleads ($\xrightarrow{=}$), Strictly Descends ($\xrightarrow{<}$), Descends ($\xrightarrow{\leq}$)) The pair (s,s') of program valuations lies in the *intraleads* relation if a configuration (s,γ) can go to the configuration (s',γ) (with the same stack) via a *local* transition or via the *finished* execution of a call statement.

$$\begin{array}{lll} s \xrightarrow{=} s' & \text{if} & (s,\gamma) \leadsto (s',\gamma) \ \text{ or} \\ & & (s,\gamma) \leadsto (s_1, \ell.\gamma) \leadsto (s_2,\gamma_2) \leadsto \ldots \leadsto (s_{n-1},\gamma_{n-1}) \ldots \leadsto (s_n, \ell.\gamma) \leadsto (s',\gamma) \\ & & \text{where } \ \gamma \in \mathsf{Loc}^\star, \ \ell \in \mathsf{Loc}, \text{ and } \gamma_2, \ldots, \gamma_{n-1} \text{ contain } \ell.\gamma \text{ as suffix} \end{array}$$

The pair (s,s') of program valuations lies in the *strictly descends* relation if a configuration (s,γ) can go to a configuration $(s', \ell.\gamma)$ via a *call* transition.

$$\begin{array}{lll} s \xrightarrow{<} s' & \text{if} & (s,\gamma) \leadsto (s', \ell.\gamma) \\ & & \quad \text{where } \ \gamma \in \mathsf{Loc}^\star \text{ and } \ell \in \mathsf{Loc} \end{array}$$

The *descends* relation $\xrightarrow{\leq}$ is the union of the two relations above.

$$\xrightarrow{\leq} \ = \ \xrightarrow{=} \cup \xrightarrow{<}$$

We can now define summaries.

Definition 2 (Summary) A summary $\mathcal{S}$ is a binary relation over program valuations that contains the transitive closure of its descends relation.

$$\mathcal{S} \quad \supseteq \quad \xrightarrow{\leq}{}^{+}$$

In other words, a summary $\mathcal{S}$ contains a pair (s, s') of program valuations if there exists a computation from a configuration (s, γ) to a configuration (s', γ') such that the initial stack γ is a suffix not only of the final stack γ' but also of every intermediate stack.

Summaries as Fixpoints. The fixpoint system below[1] is a conjunction of inclusions between relations over valuations.

Fixpoint System Φ(R,S,T)

$$\begin{array}{llll}
I1 & R \supseteq & (\mathsf{pc} = \ell_1 \wedge c \wedge \mathsf{pc}' = \ell_2) & (\ell_1, c, \ell_2) \in \mathsf{Cmds} \\
I2 & T \supseteq & R \cup T \circ R & \\
I3 & R \supseteq & (\mathsf{pc} = \ell_1 \wedge c \wedge \mathsf{pc}' = \ell_2) \ \text{ if} & \\
 & T \supseteq & (\mathsf{pc} = \mathsf{entry}_j \wedge c \wedge \mathsf{pc}' = \mathsf{exit}_j) & (\ell_1, j, \ell_2) \in \mathsf{Calls} \\
I4 & S \supseteq & (\mathsf{pc} = \ell_1 \wedge \mathsf{Var}' = \mathsf{Var} \wedge \mathsf{pc}' = \mathsf{entry}_j) & (\ell_1, j, \ell_2) \in \mathsf{Calls} \\
I5 & S \supseteq & S \circ (\mathsf{pc} = \ell_1 \wedge \mathsf{Var}' = \mathsf{Var} \wedge \mathsf{pc}' = \mathsf{entry}_j) & (\ell_1, j, \ell_2) \in \mathsf{Calls} \\
I6 & S \supseteq & S \circ T \cup T \circ S &
\end{array}$$

A fixpoint is a triple (R, S, T) that satisfies all inclusions of the form $I1$ to $I6$. It can be computed by least fixpoint iteration of (an abstraction of) the operator defined by the fixpoint system. The operator induced by $I3$ takes a set of pairs of valuations, restricts it to pairs at entry and exit locations and replaces them with the corresponding pairs at call and return locations.

[1] In our notation, we identify an assertion with the relation that it denotes. We use the operator $\circ$ for relational composition. That is, for binary relations A and B,

$$A \circ B = \{(s, s'') \mid \exists s' : (s, s') \in A \wedge (s', s'') \in B\}.$$

Theorem 1 *If the three relations over program valuations R, S and T form a fixpoint for the fixpoint system Φ, their union $\mathcal{S} = R \cup T \cup S$ is a summary for the program.*

The theorem follows from Lemmas 1 and 2 below.

Lemma 1 *The relation T is a superset of the transitive closure of the intraleads relation.*

$$T \quad \supseteq \quad \xrightarrow{=}^{+} \tag{1}$$

Proof: It is sufficient to show the statement below, which refers to configurations whose stack is empty.

If (s', ε) is $\rightsquigarrow$-reachable from (s, ε), then T contains (s, s').

We proceed by induction over the computation that leads from (s, ε) to (s', ε).

Base Step $(s, \varepsilon) \rightsquigarrow (s', \varepsilon)$

The only one-step transition that does not change the stack is a *local* transition, i.e. the valuation (s, s') satisfies an assertion of the form $\mathsf{pc} = \ell_1 \wedge \mathsf{pc}' = \ell_2 \wedge c$ where (ℓ_1, c, ℓ_2) is a command in Cmds. By inclusions $I1$ and $I2$, R and thus also T contains (s, s').

Induction Step $(s, \varepsilon) \rightsquigarrow (s_1, \gamma_1) \rightsquigarrow \ldots \rightsquigarrow (s_n, \gamma_n) \rightsquigarrow (s', \varepsilon)$**.**

Case 1. The computation from (s, ε) to (s', ε) contains no intermediate configuration with empty stack.

The stack γ_1 of the second configuration consists of one location ℓ_1, i.e. $\gamma_1 = \ell_1$, and it is equal to the stack γ_n of the last but one configuration.

The transition $(s, \varepsilon) \rightsquigarrow (s_1, \ell_1)$ is a call transition induced by, say, the call (ℓ_1, k, ℓ_2). This means that the value of the program counter in s_1 is the entry location entry_k of the called module $\mathcal{M}_k$.

The transition $(s_n, \ell_1) \rightsquigarrow (s', \varepsilon)$ is a return transition. This means that the value of the program counter in s_n is the exit location exit_k of the called module $\mathcal{M}_k$.

The computation from (s_1, ℓ_1) to (s_n, ℓ_1) is an execution (in $\mathcal{M}_k$) from entry_k to exit_k. Since no intermediate configuration has an empty stack, every intermediate stack has ℓ_1 as its first element. Hence (s_n, ε) is $\rightsquigarrow$-reachable from (s_1, ε). By induction hypothesis, T contains the pair (s_1, s_n). By inclusions $I2$ and $I3$, R and thus also T contain (s, s').

Case 2. The computation from (s,ε) to (s',ε) contains at least one intermediate configuration with empty stack.

We consider the subsequence of all configurations with empty stack in the computation.

$$(s,\varepsilon) \rightsquigarrow^+ (s_{i_1},\varepsilon) \rightsquigarrow^+ \dots \rightsquigarrow^+ (s_{i_m},\varepsilon) \rightsquigarrow^+ (s',\varepsilon)$$

For each part of the computation from (s_{i_i},ε) to $(s_{i_{i+1}},\varepsilon)$, we can apply the first case (none of the intermediate configurations has an empty stack) and obtain that R contains all pairs of valuations in consecutive configurations of the subsequence. By inclusion $I2$, T is the transitive closure of R and thus contains (s,s').

□

The proof of Lemma 1 exhibits that R is a superset of the intraleads relation.

$$R \quad \supseteq \quad \xrightarrow{=} \tag{2}$$

Since $T \supseteq R^+$ holds by $I2$, inclusion (1) is a direct consequence of inclusion (2). It seems, however, impossible to show (2) without showing (1).

Lemma 2 *The relation S is a superset of the transitive closure of the descends relation minus the transitive closure of the intraleads relation.*

$$S \quad \supseteq \quad \xrightarrow{\leq}^+ \setminus \xrightarrow{=}^+$$

Proof: Since

$$\xrightarrow{\leq}^+ \setminus \xrightarrow{=}^+ \quad = \quad (\xrightarrow{=}^\star \circ \xrightarrow{<} \circ \xrightarrow{=}^\star)^+$$

it is sufficient to show the statement below, which refers to configurations whose stack is empty.

If (s',γ') with non-empty stack γ' is $\rightsquigarrow$ -reachable from (s,ε), then S contains (s,s').

We proceed by induction over the size d of γ'.

Base Step ($d=1$) The computation leading from (s,ε) to (s',γ') is of the form

$$(s,\varepsilon) \rightsquigarrow^* (s_1,\varepsilon) \rightsquigarrow (s_2,\ell) \rightsquigarrow^* (s',\ell).$$

The transition $(s_1,\varepsilon) \rightsquigarrow (s_2,\ell)$ is a call transition. By inclusion $I4$, S contains (s_1,s_2). If s is different from s_1 or s' is different from s_2: by Lemma 1, T contains (s,s_1) resp. (s_2,s'), and by inclusion $I6$, S contains (s,s').

Induction Step ($d \Rightarrow d+1$) The computation is of the form

$$(s,\varepsilon) \rightsquigarrow^{+} (s_k,\gamma_k) \rightsquigarrow (s_{k+1},\ell.\gamma_k) \rightsquigarrow^{*} (s',\ell.\gamma_k).$$

By induction hypothesis, S contains (s,s_k). The transition from (s_k,γ_k) to $(s_{k+1},\ell.\gamma_k)$ is a call transition. By inclusion $I5$ of the fixpoint system, S contains (s_1,s_{k+1}). If s_{k+1} is different from s': by Lemma 1, T contains (s_{k+1},s), and by inclusion $I6$, S contains (s,s').

$\square$

6 Total Correctness

We assume that the correctness of the program is specified by the pair of pre- and postconditions pre and post where pre is an assertion over the set Var of unprimed program variables and post is an assertion over the set $\mathsf{Var} \cup \mathsf{Var}'$ of primed and unprimed program variables. The assertions are associated with the entry and exit points of the 'main' module M_0.

Partial correctness is the following property: if a computation starts in a configuration $q = (s,\varepsilon)$ with the empty stack and the valuation s satisfying the assertion $\mathsf{pc} = \mathsf{entry}_0 \land \mathsf{pre}$ and terminates in a configuration $q' = (s',\varepsilon)$ with the empty stack and the valuation s' satisfying the assertion $\mathsf{pc} = \mathsf{entry}_0$, then the pair of valuations (s,s') satisfies the assertion post.

Theorem 2 *The program is partially correct if and only if there exists a summary $\mathcal{S}$ whose restriction to the precondition and the entry and exit points of the 'main' module M_0 entails the postcondition.*

$$\mathcal{S} \land \mathsf{pre} \land \mathsf{pc} = \mathsf{entry}_0 \land \mathsf{pc}' = \mathsf{exit}_0 \models \mathsf{post}$$

Proof: **if-direction:** Assume that there exists a summary $\mathcal{S}$ for the program that fulfills the condition of the theorem, but the program is not partially correct. I.e. there exists a computation from an initial configuration (s,ε) that terminates in a configuration (s',ε) such that s satisfies the precondition of M_0 but (s,s') does not fulfill the postcondition.

Since $\mathcal{S}$ is a summary for the program, reachability of (s',ε) from (s,ε) wrt. $\rightsquigarrow^{+}$ implies that that (s,s') is in $\mathcal{S}$. But this is a contradiction, since $\mathcal{S}$ implies the postcondition of M_0.

only if-direction: We define $\mathcal{S}$ as the conjunction of the following relations R,T and S:

$$R = \xrightarrow{\leq} \cap \ (Acc \times Acc)\ ,$$

$$\begin{aligned} T &= \xrightarrow{\leq}^{+} \cap \ (Acc \times Acc) \ , \\ S &= \xrightarrow{<}^{+} \cap \ (Acc \times Acc) \ , \end{aligned}$$

where Acc denotes the set of all accessible states. Clearly, $\mathcal{S}$ is a summary of the program.

If the program is partially correct, each execution from an initial configuration (s, ε) to a configuration (s', ε) on termination where s fulfills the precondition pre implies that (s, s') satisfies the postcondition post; furthermore, the pair (s, s') is in T and thus in $\mathcal{S}$. This means that $\mathcal{S}$ satisfies the condition of the theorem.

$\square$

In the formulation above, the only-if direction of the theorem requires an assumption on the program syntax, namely that the 'main' module M_0 does not get called, i.e. no call is of the form $(\ell_1, 0, \ell_2)$. The assumption can always be made fulfilled by a small syntactic transformation of the program.

To see why the assumption is needed, consider the example program factorial which, in the syntax given in Section 3, does not satisfy the assumption. The S-assertion $S2$ (which refers to the precondition and the entry and exit points of the 'main' module M_0) does *not* entail the postcondition $y' = x!$ and neither does the refinement of $S2$ of the form

$$\exists n > 0 : \ \mathsf{pc} = \mathsf{entry}_0 \wedge x > 0 \wedge x' = x - n \wedge y' = (x-n)!y \wedge \mathsf{pc}' = \mathsf{exit}_0$$

which is contained in every summary of the program.

The assumption on the program syntax is not required in the formulation of the corollary below, which refers to the relation T.

Corollary 1 *The program is partially correct if and only if there exists a relation T over program valuations that is a solution in the fixpoint system Φ and whose restriction of T to the precondition and the entry and exit points of the 'main' module entails the postcondition.*

$$T \wedge \mathsf{pre} \wedge \mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pc}' = \mathsf{exit}_0 \ \models \ \mathsf{post}$$

Obviously only the inclusions of the form $I1 - I3$ of Φ are relevant for a solution for T.

Termination is the property that every computation of the program, i.e. every sequence of configurations $q_0 \rightsquigarrow q_1 \rightsquigarrow q_2 \ldots$ is finite. The next theorem states that one can characterize termination in terms of summaries.

Theorem 3 *The program is terminating if and only if there exists a summary $\mathcal{S}$ that is a finite union of well-founded relations.*

Proof: if-direction: For a proof by contradiction, we assume that there exists an infinite computation $(s_0,\varepsilon),(s_1,\gamma_1),(s_2,\gamma_2),\ldots$ starting in the empty stack. We now construct an infinite subsequence of configurations $(s^0,\gamma^0),(s^1,\gamma^1),(s^2,\gamma^2),\ldots$ such that the corresponding valuations form a descending sequence.

$$s^0 \xrightarrow{\leq} s^1 \xrightarrow{\leq} s^2 \xrightarrow{\leq} \ldots$$

The first part of the subsequence of configurations consists of all configurations with an empty stack, i.e. $(s^k,\gamma^k) = (s_{i_k},\varepsilon)$. If there are infinitely many configurations with empty stacks, then we are done with the construction and we obtain an infinite intraleads sequence.

Otherwise, there is a configuration (s_{i_k},ε) such that the stack of all subsequent configurations is not empty.

The transition from (s_{i_k},ε) to (s_{i_k+1},ℓ) is a call transition. Hence the pair of valuations (s_{i_k},s_{i_k+1}) is in $\xrightarrow{<}$.

We repeat the above construction step with (s_{i_k+1},ℓ) instead of (s_0,ε). Inductively we get an infinite sequence $s^0,s^1,s^2,\ldots$ of valuations such that pairs of consecutive valuations are in $\xrightarrow{\leq}$ and hence in $\mathcal{S}$.

We now use the assumption that $\mathcal{S}$ is a finite union of well-founded relations, say[2]

$$\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_m.$$

We define a function f with finite range that maps an ordered pair of indices of elements of the sequence $s^0,s^1,s^2\ldots$ to the index j of the relation $\mathcal{S}_j$ that contains the corresponding pair of valuations.

$$f(k,l) \overset{\text{def.}}{=} j \quad \text{where } (s^k,s^l) \in \mathcal{S}_j$$

The function f induces an equivalence relation $\sim$ on pairs of indices of $s^0,s^1,s^2,\ldots$.

$$(k_1,l_1) \sim (k_2,l_2) \quad \overset{\text{def.}}{\Leftrightarrow} \quad f(k_1,l_1) = f(k_2,l_2).$$

The index of $\sim$ is finite since the range of f is finite. By Ramsey's theorem [17], there exists an infinite set of indices K such that all pairs from K belong to the same equivalence class. Thus, there exists m and n in K, with $m<n$, such that for every k and l in K, with $k<l$, we have

[2]The assumption implies that one of the relations $\mathcal{S}_j$ occurs infinitely often in the sequence $s^0,s^1,s^2,\ldots$. This is, however, not yet a contradiction to the well-foundedness of $\mathcal{S}_j$, which needs a consecutive $\mathcal{S}_j$ -sequence.

$(k,l) \sim (m,n)$. Let $k_1, k_2, \ldots$ be the ascending sequence of elements of K. Hence, for the infinite sequence $s^{k_1}, s^{k_2}, \ldots$ we have $(s^{k_1}, s^{k_i}) \in \mathcal{S}_j$ for all $i \geq 1$. But this is a contradiction to the fact that $\mathcal{S}_j$ is well-founded.

only if-direction: Let $\mathcal{S}$ be the summary defined in the proof of Theorem 2. Assume that $\mathcal{S}$ is not a union of well-founded relations and let ς be the subrelation of $\mathcal{S}$ that is not well-founded. This means that there exists an infinite sequence $s^1, s^2, \ldots$ such that (s^i, s^{i+1}) is in ς for all $i \geq 1$.

Since s^1 is accessible, and for all $i \geq 1$ there is a non-empty computation sequence from (s^i, γ^i) to (s^{i+1}, γ^{i+1}) wrt. $\rightsquigarrow$, there exists an infinite computation $(s_1, \varepsilon), \ldots, (s^1, \gamma^1), \ldots, (s^2, \gamma^2), \ldots$ of the program. This is a contradiction to our assumption that the program is terminating. $\square$

Corollary 2 *The program is terminating if and only if there exist three relations over program valuations R, S and T that form a solution of the of the fixpoint system Φ and that are finite unions of well-founded relations.*

Deductive Verification Below we give a proof rule for the total correctness of general while programs with recursion. The proof rule is sound and complete by Theorem 1 and Corollaries 1 and 2.

Deductive verification according to the proof rule proceeds in three steps, for three given relations R, S and T over program valuations. The first step checks that the triple (R, S, T) is a fixpoint, i.e. that the relations R, S and T satisfy the inclusions given under $I1 - I6$ of the fixpoint system of Section 5. The second step checks that the restriction of the relation T to the precondition and the entry and exit points of the 'main' module entails the postcondition. The third step checks that $R \cup S \cup T$ is a finite union of well-founded relations.

$\mathcal{P}$:	program
R, T, S :	assertions over pairs of valuations
$\mathsf{pre}, \mathsf{post}$:	pre- and postconditions for $\mathcal{P}$

1. R, S and T form a fixpoint of Φ.
2. $T \wedge \mathsf{pre} \wedge \mathsf{pc} = \mathsf{entry}_0 \wedge \mathsf{pc}' = \mathsf{exit}_0 \models \mathsf{post}$
3. T and S are finite unions of well-founded relations.

Total correctness of $\mathcal{P}$: $\{\mathsf{pre}\}\ \mathcal{P}\ \{\mathsf{post}\}$

An informal description of an application of the above proof rule has been given in Section 3. It is now straightforward to instantiate the proof rule also formally for the presented examples.

Automatic Verification The inclusions $I1 - I6$ of the fixpoint system and the condition for partial correctness amounts to checking entailment between assertions. Checking the well-foundedness of the finitely many member-relations of S and T can be established automatically in many cases; see [13, 21, 12, 7]. The synthesis of the relations R, S and T is possible by least fixpoint iteration (over the domain of relations over program valuations) in combination with abstract interpretation methods [8, 9].

7 Conclusion

We have introduced a generalization of the fundamental notion of procedure summaries. Our summaries refer to arbitrarily precise descriptions of the effect of computations between general pairs of program points of general while programs (over in general infinite data domains). We have shown how one can put them to work for the verification of termination and total correctness of general while programs with recursion.

We have presented a proof rule for total correctness that uses summaries as the auxiliary assertion on the program. As already mentioned, the proof rule has an obvious potential for automation via abstract interpretation. We believe that our paper may lead to several directions of follow-up work to realize this potential, with a choice of abstraction methods (see e.g. [8, 9, 15, 11]) and techniques for the efficient construction of summaries (see e.g. [18, 2]). Other lines of future work are the extension to concurrent threads (see e.g. [10, 16]) and the account of correctness properties expressed in the CaRet logic [1].

References

[1] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of TACAS'04*, 2004.

[2] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proceedings of CAV'00*, 2000.

[3] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of PLDI'2001*, 2001.

[4] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. *In SPIN Workshop on Model Checking of Software*, 2000.

[5] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestrcited hierachical state machines. In *Proceedings of ICALP 2001*, 2001.

[6] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *In Proceedings of POPL'03*.

[7] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *Proceedings of TACAS'01*, 2001.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of POPL'77*, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL'1979*, 1979.

[10] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN03: SPIN Workshop*. Spiegel-Verlag, 2003.

[11] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *In Proceedings of SAS'04*, 2004.

[12] D. A. McAllester and K. Arkoudas. Walther recursion. In *CADE'96*, 1996.

[13] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proceedings of VMCAI'04*, 2004.

[14] A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings of LICS'04*, 2004.

[15] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Proceedings of POPL'05*, 2005.

[16] S. Qadeer, S.Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of POPL'04*, 2004.

[17] F. P. Ramsey. On a problem of formal logic. In *Proceedings London Math. Soc.*, 1930.

[18] T. Reps, M. Sagiv, and S. Horwitz. Precise interprocedural dataflow analysis via graph reachability. *Proceedings of POPL'95*, 1995.

[19] T.W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Proceedings of SAS'03*, 2003.

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[21] A. Tiwari. Termination of linear programs. In *Proceedings of CAV'04*, 2004.

[22] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*, 1986.