Nested Interpolants

Matthias Heizmann Jochen Hoenicke Andreas Podelski

University of Freiburg, Germany

Abstract

In this paper, we explore the potential of the theory of nested words for partial correctness proofs of recursive programs. Our conceptual contribution is a simple framework that allows us to shine a new light on classical concepts such as Floyd/Hoare proofs and predicate abstraction in the context of recursive programs. Our technical contribution is an interpolant-based software model checking method for recursive programs. The method avoids the costly construction of the abstract transformer by constructing a nested word automaton from an inductive sequence of 'nested interpolants' (i.e., interpolants for a nested word which represents an infeasible error trace).

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification.

Keywords Nested words, interpolants, recursion, static analysis, abstract interpretation, software model checking, abstraction refinement, Floyd-Hoare logic.

1. Introduction

A recent trend in software verification tools in the line of [5, 8, 9, 12, 14, 15, 16, 19, 20] is to use *interpolants* in order to avoid the construction of the abstract transformer for the underlying program analysis. The idea is to generate an *inductive* sequence of interpolants from a false positive (a 'spurious error trace') returned by the program analysis. Roughly, the inductiveness expresses that the interpolants can be used to form a sequence of Hoare triples which proves the infeasibility of the spurious error trace. The inductiveness thus entails the elimination of the false positive. While the approach is incarnated with great success in the 'lazy abstraction with interpolants' scheme for non-procedural programs [19], it is not clear how one can extend it to a principled method for the general class of programs with procedures, possibly recursive ones. What we are interested in here is a general foundation that allows us to systematically reason about interpolant-based correctness proofs for recursive programs. In this paper, we propose to base such a foundation on the theory of nested words [2, 3].

The motivation to use nested words stems directly from the observation that the stack-based semantics of recursive programs is not appropriate for our purpose. If we model a state as a stack of valuations of program variables and generate interpolants from a trace modeled as a sequence of states, then the interpolants cannot be restricted to contain variables of the local calling context. This defeats the very purpose of interpolants (to carry a minimal amount of information needed locally at a given position in the trace).

The theory of nested words (and nested word automata) offers an interesting potential as an alternative to the low-level view of a recursive program as a stack-based device that defines a set of traces. A nested word expresses not only the linear order of a trace but also the nesting of calls and returns. A nested word automaton does not need a stack. If we model the execution of a recursive program as a nested word, we can reason directly on the nesting, instead of having to recover the nesting with the help of a stack. We thus trade an operationally complex device defining a set of simple objects (traces) for an operationally simple device defining a set of rich objects (nested traces). The insight put forward in this work is that the increase of the object complexity is irrelevant since we reduce the reasoning of program correctness to the emptiness of the defined set of objects (the objects are existentially quantified and thus 'projected out').

In this paper, we explore the potential of the theory of nested words as a foundation for correctness proofs for the general class of recursive procedures. Our conceptual contribution is a simple framework that allows us to shine a new light on classical concepts such as Floyd/Hoare proofs and predicate abstraction for recursive programs. Our technical contribution is to give, to our knowledge for the first time, a principled method that constructs an abstract proof for recursive programs from interpolants (avoiding the construction of the abstract transformer). The method constructs a nested word automaton from an inductive sequence of 'nested interpolants' (i.e., interpolants for the 'nested trace' of the recursive program).

Related Work. Given the huge body of work on the treatment of recursion and the use of interpolants in software verification, we will discuss only the most related work.

We pick up the line of work on using nested words to provide a basis for software verification [2, 3]. A basic observation in [2, 3] is that the set of nested words generated by an abstract recursive program is definable by a finite nested word automaton. This observation refers to the setting where one has already constructed an abstract recursive program (from the program to be verified). This construction amounts to the construction of the abstract transformer, a costly step that current research tries to optimize, or to avoid altogether [5, 7, 8, 9, 12, 14, 15, 16, 19, 20] . Our work refers to the general setting of software model checking, where this step is not or not yet performed. I.e., we investigate different ways to construct a finite nested word automaton directly from the program to be verified. This allows us to give a foundation for interpolant-based proofs for recursive programs.

We distinguish two methods to use interpolants for software verification. The first method, as used, e.g., in [13, 19], generates an

```
       procedure m(x) returns (res)
       {$\top$}
$\ell_0$:  if x>100
          {$x \geq 101$}
$\ell_1$:     res:=x-10
       else
          {$x \leq 100$}
$\ell_2$:     x$_m$ := x+11
          {$x_m \leq 111$}
$\ell_3$:     call m
          {$res_m \leq 101$}
$\ell_4$:     x$_m$ := res$_m$
          {$x_m \leq 101$}
$\ell_5$:     call m
          {$res_m = 91$}
$\ell_6$:     res := res$_m$
       {$res = 91 \lor (x \geq 101 \land res = x - 10)$}
$\ell_7$:  assert (x<=101 -> res=91)
       return m
```

Figure 1. McCarthy's 91 function together with a Floyd-Hoare style annotation with invariant assertions. The program is correct if the assert statement never fails.

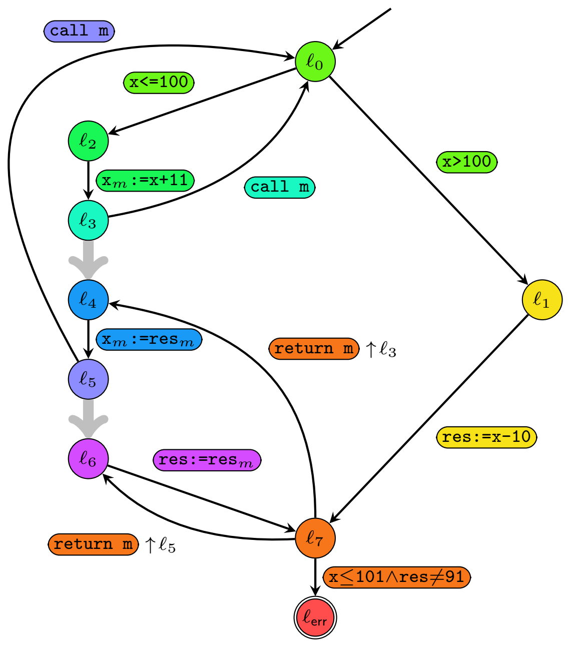


Figure 2. Recursive control flow graph for the recursive program $\mathcal{P}^{91}$ implementing McCarthy's 91 function. The initial location is ℓ_0. The error location ℓ_{err} is used to encode the correctness of the program. In Section 3 we will introduce a different reading of the graph, namely as a particular nested word automaton (over the alphabet of statements st), the *control automaton* $\mathcal{A}_{\mathcal{P}^{91}}$.

inductive sequence of interpolants from the spurious error trace of a non-recursive (non-procedural) program. The interpolants form the labels in the abstract reachability tree (in conjunctions in [19], and in tuples in [13]). The method thus constructs an interpolant-based proof and avoids the construction of an abstract transformer.

The second method, as used, e.g., in [11, 14, 17], generates a *not necessarily inductive* sequence of interpolants from the spurious error trace of a recursive program. The interpolants are used solely for defining new predicates (by subformulas). The method does construct an abstract transformer. The proof is constructed by fixpoint iteration. The iterates (i.e., not the interpolants) form a sequence of Hoare triples which proves the infeasibility of the spurious error trace. I.e., the inductiveness of sequences of interpolants is irrelevant in [11, 14].

To summarize, the first method does not account for recursion, and the second one does not consider inductive sequences of interpolants (or other ways to avoid the construction of the abstract transformer). In contrast, we use nested words to formalize inductive sequences of interpolants for traces of recursive programs (and nested word automata to avoid the construction of the abstract transformer).

2. Preliminaries

In this section, we fix the notation for recursive programs (following [21]) and for nested words (following [2, 3]).

2.1 Recursive Programs

Informal Presentation. We consider a simple procedural programming language in a BoogiePL-like syntax [6]. As usual, we impose a number of restrictions and conventions in order to simplify the formal exposition. The language has no pointers, no global variables and only call-by-value procedure calls. It is forbidden to write to the input variables of a procedure. A procedure p has one input parameter x and one output parameter res. Procedure calls appear in the following form.

$$res_p \texttt{ := call p(}x_p\texttt{)}$$

If procedure p with formal parameter x is called, the argument (actual parameter) is a special variable x_p. On return, the value of the procedure's output variable res is stored into the special variable res_p of the caller. Before calling procedure p, the caller writes the input value to the variable x_p. The callee accesses the value through the variable x. The callee stores the result value into the variable res. The result value is returned to the caller through the variable res_p. Procedures with arbitrarily many input and output parameters can be treated in an analogous way. A global variable can be modeled by adding an input and an output variable to each procedure that directly or indirectly (via a recursive function call) accesses the global variable.

Figure 1 shows an implementation of McCarthy's 91 function (if the argument x is not greater than 101, the function returns 91) in a code fragment that meets our restrictions and conventions.

Formal Presentation: Recursive Control Flow Graph. Following [21], we present a recursive program formally as a *recursive control flow graph*; see Figure 2 for an example. Each node is a program location ℓ. Each edge is labeled with a statement st, which is either an assignment `y:=t`, an assume ϕ, a call `call p`, or a return `return p`.

For each procedure p, the recursive control flow graph contains p's control flow graph. The edges between internal nodes are labeled with assignment or assume statements. The nodes of p's control flow graph are the program locations of p, say $\ell_0^p, \ldots, \ell_n^p$ where the node ℓ_0^p is the entry location of p and ℓ_n^p its exit location. To model that procedure p is called from procedure q at program location ℓ_j^q (in q's control flow graph),

- an edge labeled `call p` goes from ℓ_j^q to ℓ_0^p, the entry location of p,

	state	label of edge (ℓ, ℓ')	successor state	side condition
assignment	$S.(\ell, \nu)$	`y:=t`	$S.(\ell', \nu')$	$\nu' = \nu \oplus \{y \mapsto \nu(t)\}$
assume	$S.(\ell, \nu)$	$\boxed{\phi}$	$S.(\ell', \nu)$	$\nu \models \phi$
call	$S.(\ell, \nu)$	`call p`	$S.(\ell, \nu).(\ell', \nu')$	$\nu'(x) = \nu(x_p)$
return	$S.(\ell_<, \nu_<).(\ell, \nu)$	`return p` $\uparrow \ell_<$	$S.(\ell', \nu')$	$\nu' = \nu_< \oplus \{res_p \mapsto \nu(res)\}$

Figure 3. Semantics of statements in the context of a recursive control flow graph (where a statement is used to label an edge between two nodes ℓ and ℓ'). A state is a stack of local states/location-valuation pairs; the current local state is the topmost/rightmost one.

- an edge labeled `return p` $\uparrow \ell_j^q$ goes from ℓ_n^p, the exit location of p, to the location ℓ_{j+1}^q in q's control flow graph, where ℓ_{j+1}^q is the successor location of ℓ_j^q with respect to the call of procedure p.

In Figure 2, we use a thick gray edge in order to depict the successor location of a call location, with respect to the procedure calls initiated by `call m`. The thick gray edges are not edges of the recursive control flow graph. The assert statement from Figure 1 is translated to an edge which is labeled with an assume statement and leads to a special error location.

Semantics. A *valuation* ν is a function that maps program variables to values. A (local) state of a procedure is a pair (ℓ, ν), consisting of a program location and a valuation. A state S of the (whole) program is a stack of local states, i.e., location-valuation pairs, which we write as a sequence

$$S = (\ell_0, \nu_0).(\ell_1, \nu_1) \dots (\ell_n, \nu_n).$$

Each element corresponds to a called (and not yet returned) procedure. The topmost/rightmost element represents the actual calling context.

Each statement st induces a transition relation between states; see Figure 3. We use the usual notation for a transition from a state S to a successor state S'.

$$S \xrightarrow{st} S'$$

Traces π***.*** A *trace* π is a sequence of statements, $\pi = st_0 \dots st_{n-1}$. We extend the transition relation from single statements to traces in the usual way and use the same notation also for traces.

$$S \xrightarrow{\pi} S'$$

A trace π is a *feasible error trace* of the program $\mathcal{P}$ if there exists an initial valuation ν_0, a stack of local states S and a final valuation ν_n such that the triple

$$(\ell_0, \nu_0) \xrightarrow{\pi} S.(\ell_{\mathsf{err}}, \nu_n)$$

is a transition of the program (where ℓ_0 is the initial location of the main procedure of $\mathcal{P}$ and ℓ_{err} is an error location of $\mathcal{P}$).

Program Correctness = No Feasible Error Traces. The recursive control flow graph encodes program correctness through special nodes corresponding to *error locations*. We define the correctness of the program through the non-reachability of an error location ℓ_{err} by a program execution. Formally, a program $\mathcal{P}$ is *correct* if it has no feasible error trace.

The classical notion of partial correctness can be accommodated by representing precondition and postcondition by assume statements and assert statements (the latter is translated to an edge that leads to the error location).

2.2 Nested Words

Following [2, 3], a *nested word* over an alphabet Σ is a pair $(w, \rightsquigarrow)$ consisting of a word $w = a_0 \dots a_{n-1}$ over the alphabet Σ and the *nesting relation* $\rightsquigarrow$. The fact that $\rightsquigarrow$ is a nesting relation for w means that we have a relation between the n positions of w, formally

$$\rightsquigarrow \subseteq \{0, \dots, n-1\} \times \{0, \dots, n-1, \infty\}$$

which is left-unique, right-unique, and properly nested, formally:

$$i_1 \rightsquigarrow j,\ i_2 \rightsquigarrow j,\ j \neq \infty \text{ implies } i_1 = i_2$$

$$i \rightsquigarrow j_1,\ i \rightsquigarrow j_2 \text{ implies } j_1 = j_2$$

$$i_1 \rightsquigarrow j_1,\ i_2 \rightsquigarrow j_2,\ i_1 \leq i_2 \text{ implies } \begin{cases} i_1 < j_1 < i_2 < j_2 \\ \text{or} \\ i_1 \leq i_2 < j_2 \leq j_1. \end{cases}$$

The idea is that the relation between the position i of a call and the position j of a matching return is expressed by $i \rightsquigarrow j$. Positions appearing on the left (resp. right) in pairs in $\rightsquigarrow$ are called *call* (resp. *return positions*). All other positions are *internal positions*. The index ∞ is used as return position for all unfinished calls. For a return position $j \neq \infty$, the corresponding call position (i.e., the unique position i such that $i \rightsquigarrow j$) is the *call predecessor* of j. In contrast with [3], we do not allow $-\infty$ as a call predecessor. In contrast with [2], we do allow ∞ as a return position.

Possibly Infinite Nested Word Automata (NWA). A *nested word automaton* over an alphabet Σ is a tuple

$$\mathcal{A} = (Q, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle, Q^{\mathsf{init}}, Q^{\mathsf{fin}})$$

consisting of

- a (not necessarily finite) set of states Q,
- a triple $\langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle$ of transition relations for, respectively, internal, call, and return positions,

$$\begin{aligned} \delta_{\mathsf{in}} &\subseteq Q \times \Sigma \times Q \\ \delta_{\mathsf{ca}} &\subseteq Q \times \Sigma \times Q \\ \delta_{\mathsf{re}} &\subseteq Q \times Q \times \Sigma \times Q \end{aligned}$$

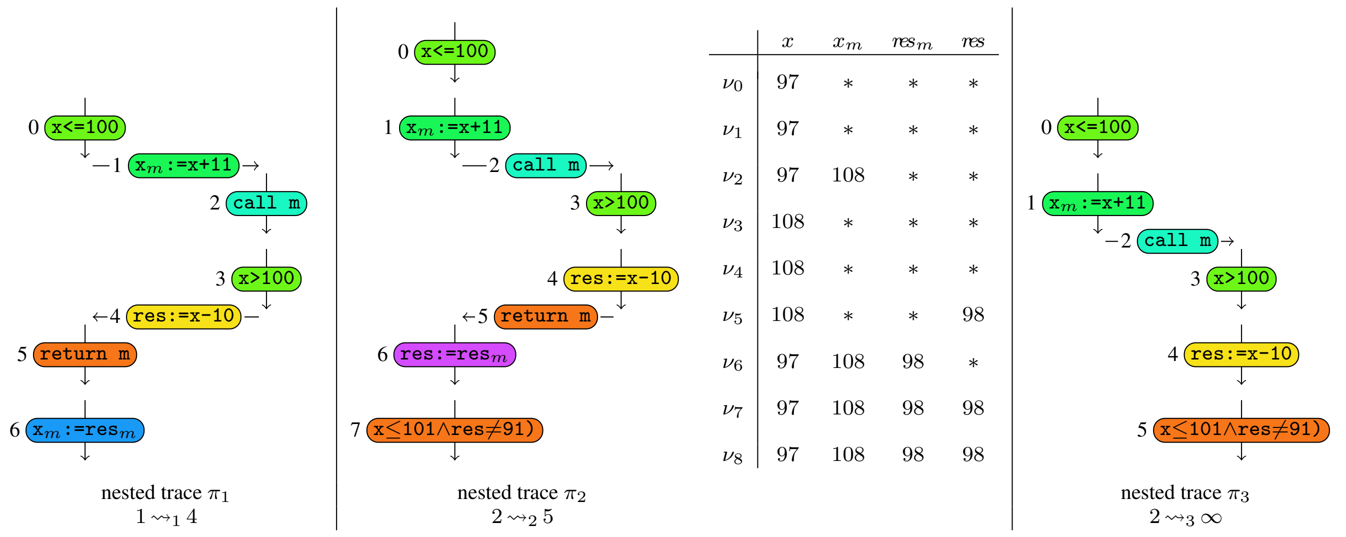
- a set of initial states $Q^{\mathsf{init}} \subseteq Q$,
- a set of final state $Q^{\mathsf{fin}} \subseteq Q$.

A *run* of a nested word automaton $\mathcal{A}$ over the nested word $(a_0 \dots a_{n-1}, \rightsquigarrow)$ is a sequence $q_0, \dots, q_n$ of states that starts in an initial state, i.e., $q_0 \in Q^{\mathsf{init}}$, and that is consecutive, i.e., for each $i = 0, \dots, n-1$,

$$\begin{aligned} (q_i, a_i, q_{i+1}) &\in \delta_{\mathsf{in}} && \text{if } i \text{ is an internal position,} \\ (q_i, a_i, q_{i+1}) &\in \delta_{\mathsf{ca}} && \text{if } i \text{ is a call position,} \\ (q_i, q_k, a_i, q_{i+1}) &\in \delta_{\mathsf{re}} && \text{if } i \text{ is a return position and } k \rightsquigarrow i. \end{aligned}$$

The run is *accepting* if it ends in a final state, i.e., $q_n \in Q^{\mathsf{fin}}$.

The nested word automaton $\mathcal{A}$ *accepts* the nested word $(w, \rightsquigarrow)$ if it has an accepting run over $(w, \rightsquigarrow)$. The language of nested words recognized by $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A})$ consisting of the nested words accepted by $\mathcal{A}$.



	x	x_m	res_m	res
ν_0	97	$*$	$*$	$*$
ν_1	97	$*$	$*$	$*$
ν_2	97	108	$*$	$*$
ν_3	108	$*$	$*$	$*$
ν_4	108	$*$	$*$	$*$
ν_5	108	$*$	$*$	98
ν_6	97	108	98	$*$
ν_7	97	108	98	98
ν_8	97	108	98	98

Figure 4. Examples of nested traces (over the alphabet of statements used in the recursive program $\mathcal{P}^{91}$ implementing McCarthy's 91 function). The indentation of the statements indicates the nesting relation. The nested trace π_3 is accepted by the control automaton $\mathcal{A}_{\mathcal{P}^{91}}$; the two others are not. The nested trace π_2 is accepted by the data automaton $\mathcal{A}_\Sigma$ (the sequence $\nu_0, \dots, \nu_8$ is an accepting run); the two others are not.

Deviating from [2, 3], we do not require that the set of states of a nested word automaton is finite. In the next section, we will define the *data automaton* as a possibly infinite nested word automaton. All other automata considered in this presentation are finite.

The notion of regularity does not deviate from [2, 3]. A language of nested words is *regular* if it is recognized by a *finite* nested word automaton. Regular languages of nested words enjoy the standard properties of regular language theory, of which we will use the closure under intersection and complement, and the decidability of emptiness [2, 3].

3. An NWA View of Program Correctness

The first step to formulate a view of program correctness in terms of nested words is to fix the alphabet, namely as the set of statements. Once we have done that it is very natural to give a view of Program correctness, to define a new proof rule based on nested word automaton, and to give a new formulation of predicate abstraction based on nested word automata.

3.1 An NWA characterization of Program Correctness

We assume a fixed set of statements which we note Σ. We will view Σ as an alphabet, statements st as letters and traces $\mathit{st}_0 \dots \mathit{st}_{n-1}$ as words; i.e., $\mathit{st}_0 \dots \mathit{st}_{n-1} \in \Sigma^*$.

We define a *nested trace* to be a nested word

$$\pi = (\mathit{st}_0 \dots \mathit{st}_{n-1}, \rightsquigarrow)$$

over the alphabet Σ of statements.

Given a nested trace over the alphabet Σ whose letters include call and return statements, it makes sense to refine the notion of nesting. We say that $\pi = (\mathit{st}_0 \dots \mathit{st}_{n-1}, \rightsquigarrow)$ is *well-nested* if the nesting relation between its positions is consistent with the letters at the respective positions; i.e., we have

- the letter st_i is a call statement if and only if i is a call position, and
- the letter st_i is a return statement if and only if i is a return position.

Example 1. *Figure 4 depicts three examples of nested traces (over the alphabet of statements used in the McCarthy example). The three corresponding nesting relations are given below.*

$$1 \rightsquigarrow_1 4$$

$$2 \rightsquigarrow_2 5$$

$$2 \rightsquigarrow_3 \infty$$

In the figure we use indentation to indicate the nesting relation.

Nested Traces and Possible Program Executions. The notion of a nested trace does not refer to a specific program (in fact, it may not correspond to a path in the given program's recursive control flow graph), and it does not take into account the semantics of statements (it may not correspond to a possible execution of any program).

We will next introduce two properties of a nested trace which together imply the fact that it corresponds to a possible execution of the given program. The two properties characterize two orthogonal aspects ("control" vs. "data") of this fact. Each of the two properties will be presented as a nested word automaton, $\mathcal{A}_\mathcal{P}$ resp. $\mathcal{A}_\Sigma$.

The first property characterizes the fact that the nested trace complies with the control flow prescribed by the recursive program graph (and that it is well-nested). This property is a regular language of nested words; we will use the *control automaton* $\mathcal{A}_\mathcal{P}$, a *finite* nested word automaton, to present it.

The second property characterizes that the nested trace complies with the semantics of statements as manipulators of data (and that it is well-nested). This property is in general *not* a regular language of nested words; we will use the *data automaton* $\mathcal{A}_\Sigma$, an in general infinite nested word automaton, to present it.

Control. From now on, we assume a fixed program $\mathcal{P}$ given formally as a recursive control flow graph. The set of nested traces that comply with the control expressed by $\mathcal{P}$ (ignoring the data) can be formally defined by the recursive program graph viewed as a nested word automaton. I.e., the edges of the recursive program graph defines the three kinds of transition relations; the initial location defines the set of initial states; the error location defines the set of final states.

Definition 1 (Control Automaton $\mathcal{A}_\mathcal{P}$, Nested Error Trace)**.** *Given the program $\mathcal{P}$, the control automaton $\mathcal{A}_\mathcal{P}$ is the nested word automaton*

$$\mathcal{A}_\mathcal{P} = (Q, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle, Q^{\mathsf{init}}, Q^{\mathsf{fin}})$$

where

- *the set of states Q is the set of nodes (i.e., program locations),*
- *the three transition relations δ_{in}, δ_{ca}, δ_{re} are the edge relations; i.e., if the edge (ℓ, ℓ') is labeled with:*
  - *the assignment or assume statement st, then*

    $$(\ell, \mathit{st}, \ell') \in \delta_{\mathsf{in}}$$
  - *the call statement $\boxed{\texttt{call p}}$, then*

    $$(\ell, \boxed{\texttt{call p}}, \ell') \in \delta_{\mathsf{ca}}$$
  - *the return statement (with call location $\ell_<$) $\boxed{\texttt{return p}} \uparrow \ell_<$, then*

    $$(\ell, \ell_<, \boxed{\texttt{return p}}, \ell') \in \delta_{\mathsf{re}}$$
- *the set of initial states Q^{init} consists of the initial location of the main procedure,*

  $$Q^{\mathsf{init}} = \{\ell_0^{main}\}$$
- *the set of final states Q^{fin} consists of the error location,*

  $$Q^{\mathsf{fin}} = \{\ell_{\mathsf{err}}\}.$$

A nested error trace is a nested word accepted by the control automaton $\mathcal{A}_\mathcal{P}$.

An alternative, richer way to define the regular language of nested error traces uses the automata-theoretic product of two nested word automata. The first one is the recursive program graph (without error locations) viewed as a nested word automaton (just as we did for $\mathcal{A}_\mathcal{P}$) where every program location is a final state; i.e., $Q^{\mathsf{fin}} = Q$. The second one is the nested word automata that corresponds to the negation of the CaRet property that defines the program correctness [1].

Example 2. *The control automaton $\mathcal{A}_{\mathcal{P}^{91}}$ of the program $\mathcal{P}^{91}$ is given in Figure 2. In the figure, a return edge labeled by a return statement together with the call location represents a transition in δ_{re}. For example, the edge labeled $(\ell_7, \boxed{\texttt{return m}} \uparrow \ell_5, \ell_6)$ represents the transition $(\ell_7, \ell_5, \boxed{\texttt{return m}}, \ell_6)$ in δ_{re}. We next take the three nested traces from Example 1 (given in Figure 4) and investigate whether they are accepted by $\mathcal{A}_{\mathcal{P}^{91}}$.*

The nested trace π_1 is not accepted because position 1 is a call position according to the nesting relation (and hence the successor state should be given by the transition relation δ_{ca}), but the letter at position 1 is not a call statement (and hence there is no successor state according to the transition relation δ_{ca}).

The nested trace π_2 is not accepted. The run

$$r = \ell_0, \ell_2, \ell_3, \ell_0, \ell_1, \ell_7, \ell_6, \ell_7, \ell_{\mathsf{err}}$$

is possible according to the graph structure of $\mathcal{A}_{\mathcal{P}^{91}}$. However, Position 5 of π_2 is a return position, and position 2 its call predecessor. Only if $(\ell_7, \ell_3, \boxed{\texttt{return m}}, \ell_6)$ was a transition in δ_{re} (corresponding to a return edge of $\mathcal{A}_{\mathcal{P}^{91}}$), r would be a valid run.

This (the fact that the nested trace π_2 is not accepted) illustrates that the control automaton enforces the call/return discipline; i.e., return statements leading to ℓ_6 can only return from procedures called from location ℓ_5.

The nested trace π_3 is accepted by $\mathcal{A}_{\mathcal{P}^{91}}$. This illustrates the fact that the control automaton accepts traces with unmatched call statements. The example also illustrates that the existence of a nested error trace (i.e., the non-emptiness of the control automaton $\mathcal{A}_\mathcal{P}$) does not yet imply a violation of the correctness specification ("control is not enough; we also need data").

Data. We next define the data automaton $\mathcal{A}_\Sigma$. This (in general infinite) nested word automaton recognizes an (in general non-regular) set of nested traces. This set contains the well-nested traces that are possible according to the semantics of statements (whose execution manipulates data). We call these traces *feasible nested traces*.

This is the only point in the presentation where we define an infinite nested word automaton. In fact, from now on we will construct finite nested word automata that approximate the infinite data automaton $\mathcal{A}_\Sigma$ (i.e., recognize a superset).

Definition 2 (Data Automaton $\mathcal{A}_\Sigma$, Feasible Nested Trace)**.** *Given the set of statements Σ, the data automaton*

$$\mathcal{A}_\Sigma = (Q, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle, Q^{\mathsf{init}}, Q^{\mathsf{fin}})$$

is the (in general infinite) nested word automaton where

- *the set of states Q is the (in general infinite) set of valuations ν,*
- *the three transition relations δ_{in}, δ_{ca}, δ_{re} are the transition relations induced by the statements in Σ; i.e., if the statement is:*
  - *the assignment statement $\boxed{\texttt{y:=t}}$, then*

    $$(\nu, \boxed{\texttt{y:=t}}, \nu \oplus \{y \mapsto \nu(t)\}) \in \delta_{\mathsf{in}}$$
  - *the assume statement $\boxed{\phi}$, then*

    $$(\nu, \boxed{\phi}, \nu) \in \delta_{\mathsf{in}} \;\; if \;\; \nu \models \phi$$
  - *the call statement $\boxed{\texttt{call p}}$, then*

    $$(\nu, \boxed{\texttt{call p}}, \nu') \in \delta_{\mathsf{ca}} \;\; if \;\; \nu'(x) = \nu(x_p)$$
  - *the return statement $\boxed{\texttt{return p}}$, then*

    $$(\nu, \nu_<, \boxed{\texttt{return p}}, \nu_< \oplus \{\mathit{res}_p \mapsto \nu(\mathit{res})\}) \in \delta_{\mathsf{re}}\,.$$
- *every state is an initial state (i.e. $Q^{\mathsf{init}} = Q$),*
- *every state is a final state (i.e. $Q^{\mathsf{fin}} = Q$).*

A feasible nested trace is a nested word is accepted by the data automaton.

Example 3. *We take the three nested traces from Example 1 (given in Figure 4) and investigate whether they are accepted by $\mathcal{A}_\Sigma$. The nested trace π_1 is not accepted; the same explanation as with $\mathcal{A}_\mathcal{P}$ applies. In fact, both a control automaton and a data automaton accept only well-nested traces.*

The nested trace π_2 is accepted (i.e., it is a feasible nested trace). The sequence $\nu_0, \ldots, \nu_8$ is an accepting run.

The nested trace π_3 is not accepted because there is no valuation/state ν with a run that reads the three letters: the assume statement $\boxed{\texttt{x>100}}$, the assignment $\boxed{\texttt{res:=x-10}}$ and the assume statement $\boxed{\texttt{x}\leq\texttt{101}\wedge\texttt{res}\neq\texttt{91}}$.

Theorem 1 (Characterizing Program Correctness by NWA's)**.** *The program $\mathcal{P}$ is correct if and only if the intersection between the control automaton and the data automaton is empty, formally*

$$\mathcal{L}(\mathcal{A}_\mathcal{P}) \cap \mathcal{L}(\mathcal{A}_\Sigma) = \emptyset.$$

Proof. Using the three notions of: feasible error trace (defined in Section 2.1), nested error trace (i.e., accepted by $\mathcal{A}_\mathcal{P}$), and feasible nested trace (i.e., accepted by $\mathcal{A}_\Sigma$), we can rephrase the theorem as follows. A trace $\mathit{st}_0 \ldots \mathit{st}_{n-1}$ is a feasible error trace of $\mathcal{P}$ if and only if there exists a nesting relation $\rightsquigarrow$ such that the nested trace $(\mathit{st}_0 \ldots \mathit{st}_{n-1}, \rightsquigarrow)$ is at the same time a nested error trace and a feasible nested trace.

We show a more general statement by induction on the length n of the trace. For every nesting relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \{0,\dots,n-1\} \times \{0,\dots,n-1,\infty\}$, every sequence of locations $\ell_0,\dots,\ell_n$ and every sequence of valuations $\nu_0,\dots,\nu_n$ the following two statements are equivalent.

- The sequence of locations $\ell_0,\dots,\ell_n$ is a run of $\mathcal{A}_{\mathcal{P}}$ for $(st_0 \dots st_{n-1}, \rightsquigarrow)$ and the sequence of valuations $\nu_0,\dots,\nu_n$ is a run of $\mathcal{A}_{\Sigma}$ for $(st_0 \dots st_{n-1}, \rightsquigarrow)$.
- The nested trace $(st_0 \dots st_{n-1}, \rightsquigarrow)$ is well-nested and there is a sequence of stacks
$$(\ell_0,\nu_0) \xrightarrow{st_0} S_1.(\ell_1,\nu_1) \xrightarrow{st_1} \cdots \xrightarrow{st_{n-1}} S_n.(\ell_n,\nu_n)$$
according to the transition relation of $\mathcal{P}$. In that case $S_i = (\ell_{k_0},\nu_{k_0})\dots(\ell_{k_m},\nu_{k_m})$ such that the nesting relation contains $k_0 \rightsquigarrow k'_0,\dots,k_m \rightsquigarrow k'_m$ and $k_0 < \dots < k_m < i \leq k'_m \leq \dots \leq k'_0$ holds. □

3.2 Proof Rule Based on Finite NWA

The following proof rule for the correctness of recursive programs uses a finite nested word automaton $\mathcal{A}$ as a proof argument: if $\mathcal{A}$ recognizes a superset of the set of feasible nested traces and it does not intersect with the control automaton, then the program is correct.

$$\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma}),\ \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset \implies \mathcal{P} \text{ is correct} \quad (1)$$

The proof rule is interesting by the theory of regular languages of nested words; once we have constructed a finite nested word automaton $\mathcal{A}$ that approximates the data automaton, the emptiness of its intersection with the control automaton can be tested effectively [2, 3].

The soundness of the proof rule (1) follows from the only-if direction of Theorem 1. The question is: if the program $\mathcal{P}$ is correct, is there always a proof argument in the form of a *finite* nested word automaton $\mathcal{A}$? Perhaps surprisingly, the answer is yes: the proof rule is complete. Completeness notoriously holds (if it holds) by a disappointingly trivial line of reasoning with no indication on how to find the proof argument. For the proof rule (1), it is sufficient to take the complement of $\mathcal{A}_{\mathcal{P}}$ as a candidate for $\mathcal{A}$. Clearly its intersection with $\mathcal{A}_{\mathcal{P}}$ is empty. Since $\mathcal{A}_{\mathcal{P}}$ does not accept any feasible nested trace (otherwise $\mathcal{P}$ would not be correct), or: it recognizes a subset of infeasible nested traces, its complement recognizes a superset of feasible traces. We summarize our discussion by the following statement.

Theorem 2 (Soundness and Completeness of Proof Rule (1))**.** *The program $\mathcal{P}$ is correct if and only if there exists a finite nested word automaton $\mathcal{A}$ that recognizes a superset of the set of feasible nested traces (i.e., $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma})$) and that does not intersect with the control automaton (i.e., $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset$).*

In the remainder of this presentation, we will investigate systematic ways to construct candidate proof arguments for a finite nested word automaton $\mathcal{A}$.

3.3 Predicate Abstraction

In this section, we automate the proof rule given in the previous section by giving a predicate abstraction-based proof method for the correctness of a recursive program. The proof method uses predicate abstraction to construct a finite nested word automaton $\mathcal{A}_{\mathsf{Pred}}$ which recognizes a superset of the set of feasible nested traces. The proof succeeds if the intersection of $\mathcal{A}_{\mathsf{Pred}}$ with the control automaton $\mathcal{A}_{\mathcal{P}}$ is empty.

Intuitively, the proof method constructs $\mathcal{A}_{\mathsf{Pred}}$ by abstracting the data automaton $\mathcal{A}_{\Sigma}$. The abstraction consists essentially of transforming the triple of transition relations between concrete valuations (the states of $\mathcal{A}_{\Sigma}$) into a triple of transition relations between abstract valuations. Abstract valuations are defined in terms of predicates; we will formally introduce abstract valuations as *bitvectors*.

Predicates. In our context, a *predicate* is a set of valuations. Such a set may be defined by an assertion, e.g., $x \geq 101$.

Bitvectors. Given a finite set of predicates, say
$$\mathsf{Pred} = \{p_1,\dots,p_m\}$$
we call an m-tuple $\boldsymbol{b} \in \{0,1\}^m$ a *bitvector*. Assuming a fixed order on the predicates, a bitvector $\boldsymbol{b} = \langle b_1,\dots,b_m\rangle$ has a meaning defined by
$$[\![\langle b_1,\dots,b_m\rangle]\!] = \{\nu \mid \forall j \in \{1,\dots,m\}.\, \nu \in p_j \Leftrightarrow b_j = 1\}.$$

Nested Post Operators $\langle post_{\mathsf{in}}, post_{\mathsf{ca}}, post_{\mathsf{re}}\rangle$**.** The triple of nested post operators $\langle post_{\mathsf{in}}, post_{\mathsf{ca}}, post_{\mathsf{re}}\rangle$ defined in Figure 5 are in direct connection with the triple of transition relations $\langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}}\rangle$ of the data automaton $\mathcal{A}_{\Sigma}$. Namely, for a set $\mathcal{V}$ of valuations, we have:
$$\begin{aligned}
post_{\mathsf{in}}(\mathcal{V}, st) &= \{\nu' \mid \exists\, \nu \in \mathcal{V} : (\nu, st, \nu') \in \delta_{\mathsf{in}}\}\\
post_{\mathsf{ca}}(\mathcal{V}, st) &= \{\nu' \mid \exists\, \nu \in \mathcal{V} : (\nu, st, \nu') \in \delta_{\mathsf{ca}}\}\\
post_{\mathsf{re}}(\mathcal{V}, \mathcal{V}_<, st) &= \{\nu' \mid \exists\, \nu \in \mathcal{V}\, \exists\, \nu_< \in \mathcal{V}_< : \\
&\qquad \nu(x) = \nu_<(x_p),\\
&\qquad (\nu, \nu_<, st, \nu') \in \delta_{\mathsf{re}}\}
\end{aligned}$$

Abstract Nested Post Operators $\langle post^{\#}_{\mathsf{in}}, post^{\#}_{\mathsf{ca}}, post^{\#}_{\mathsf{re}}\rangle$**.** The translation of the triple $\langle post_{\mathsf{in}}, post_{\mathsf{ca}}, post_{\mathsf{re}}\rangle$ of nested post operators into the triple $\langle post^{\#}_{\mathsf{in}}, post^{\#}_{\mathsf{ca}}, post^{\#}_{\mathsf{re}}\rangle$ of abstract nested post operators (over bitvectors) is as one expects; see Figure 5.

Definition 3 (Predicate Automaton $\mathcal{A}_{\mathsf{Pred}}$)**.** *Given the set of predicates* Pred, *the* predicate automaton
$$\mathcal{A}_{\mathsf{Pred}} = (Q, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}}\rangle, Q^{\mathsf{init}}, Q^{\mathsf{fin}})$$
is the finite nested word automaton where

- *the set of states Q consists of all bitvectors,*
- *every state is initial, i.e., $Q^{\mathsf{init}} = Q$,*
- *every state is final, i.e., $Q^{\mathsf{fin}} = Q$,*
- *the triple of transition relations $\langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}}\rangle$ corresponds to the triple $\langle post^{\#}_{\mathsf{in}}, post^{\#}_{\mathsf{ca}}, post^{\#}_{\mathsf{re}}\rangle$ of abstract nested post operators; i.e.,*
  - *if st is an assignment or assume statement, then*
    $(\boldsymbol{b}, st, \boldsymbol{b}') \in \delta_{\mathsf{in}}$ *if* $\boldsymbol{b}' \in post^{\#}_{\mathsf{in}}(\boldsymbol{b}, st)$,
  - *if st is a call statement, then*
    $(\boldsymbol{b}, st, \boldsymbol{b}') \in \delta_{\mathsf{ca}}$ *if* $\boldsymbol{b}' \in post^{\#}_{\mathsf{ca}}(\boldsymbol{b}, st)$,
  - *if st is a return statement, then*
    $(\boldsymbol{b}, \boldsymbol{b}_<, st, \boldsymbol{b}') \in \delta_{\mathsf{re}}$ *if* $\boldsymbol{b}' \in post^{\#}_{\mathsf{re}}(\boldsymbol{b}, \boldsymbol{b}_<, st)$.

Theorem 3. *The predicate automaton $\mathcal{A}_{\mathsf{Pred}}$ recognizes a superset of the set of feasible nested traces, i.e., $\mathcal{L}(\mathcal{A}_{\mathsf{Pred}}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma})$.*

Proof. An accepting run $\nu_0,\dots,\nu_n$ of the data automaton $\mathcal{A}_{\Sigma}$ on a nested trace π gives rise to an accepting run $\boldsymbol{b}_0,\dots,\boldsymbol{b}_n$ of the predicate automaton $\mathcal{A}_{\mathsf{Pred}}$ on π where the i-th bitvector contains the i-th valuation, formally $\nu_i \in [\![\boldsymbol{b}_i]\!]$ for $i = 0,\dots,n$. □

$$\begin{array}{rcl}
post_{\mathsf{in}}(\mathcal{V}, \boxed{\texttt{y:=t}}) & = & \{\nu \oplus \{y \mapsto \nu(t)\} \mid \nu \in \mathcal{V}\} \\
post_{\mathsf{in}}(\mathcal{V}, \boxed{\phi}) & = & \{\nu \mid \nu \in \mathcal{V}, \nu \models \phi\} \\
post_{\mathsf{ca}}(\mathcal{V}, \boxed{\texttt{call p}}) & = & \{\nu' \mid \{x \mapsto \nu(x_p)\} \in \nu', \nu \in \mathcal{V}\} \\
post_{\mathsf{re}}(\mathcal{V}, \mathcal{V}_<, \boxed{\texttt{return p}}) & = & \{\nu_< \oplus \{res_p \mapsto \nu(res)\} \mid \nu(x) = \nu_<(x_f), \nu_< \in \mathcal{V}_<, \nu \in \mathcal{V}\} \\
post^{\#}_{\mathsf{in}}(\boldsymbol{b}, \boxed{\texttt{y:=t}}) & = & \{\boldsymbol{b}' \mid post_{\mathsf{in}}([\![\boldsymbol{b}]\!], \boxed{\texttt{y:=t}}) \cap [\![\boldsymbol{b}']\!] \neq \emptyset\} \\
post^{\#}_{\mathsf{in}}(\boldsymbol{b}, \boxed{\phi}) & = & \{\boldsymbol{b}' \mid post_{\mathsf{in}}([\![\boldsymbol{b}]\!], \boxed{\phi}) \cap [\![\boldsymbol{b}']\!] \neq \emptyset\} \\
post^{\#}_{\mathsf{ca}}(\boldsymbol{b}, \boxed{\texttt{call p}}) & = & \{\boldsymbol{b}' \mid post_{\mathsf{ca}}([\![\boldsymbol{b}]\!], \boxed{\texttt{call p}}) \cap [\![\boldsymbol{b}']\!] \neq \emptyset\} \\
post^{\#}_{\mathsf{re}}(\boldsymbol{b}, \boldsymbol{b}_<, \boxed{\texttt{return p}}) & = & \{\boldsymbol{b}' \mid post_{\mathsf{re}}([\![\boldsymbol{b}]\!], [\![\boldsymbol{b}_<]\!], \boxed{\texttt{return p}}) \cap [\![\boldsymbol{b}']\!] \neq \emptyset\}
\end{array}$$

Figure 5. The triple $\langle post_{\mathsf{in}}, post_{\mathsf{ca}}, post_{\mathsf{re}}\rangle$ of nested post operators (over sets of valuations $\mathcal{V}$) is in correspondence with the triple of transition relations of the data automaton $\mathcal{A}_\Sigma$. The triple $\langle post^{\#}_{\mathsf{in}}, post^{\#}_{\mathsf{ca}}, post^{\#}_{\mathsf{re}}\rangle$ of abstract nested post operators (over bitvectors) is in correspondence with the triple of transition relations of the predicate automaton $\mathcal{A}_{\mathsf{Pred}}$.

4. Nested Interpolant Automata

We now define a class of finite nested word automata which recognize subsets of the set of infeasible traces. That is, we will use their complement for the nested word automaton $\mathcal{A}$ that serves as proof argument in the proof rule (1) of the previous section.

In many of the settings that we consider, we have a *sequence of predicates* $I_0, I_1, \ldots, I_n$ (we refer to the predicates as interpolants for reasons that will become apparent later); this sequence is related to a nested error trace π; it may be generated, for example, by the proof of the infeasibility of π.

The general notion of an *interpolant automaton* that we introduce below, however, does not refer to an error trace. It refers to an arbitrary sequence of predicates $I_0, I_1, \ldots, I_n$. Given such a sequence, we will associate each predicate I_i with an automaton state q_i. The automaton states are not necessarily pairwise distinct; i.e., we may associate two different predicates I_i and I_j with the same automaton state, and we may associate the same predicate with two different states (i.e., we may have $I_i \neq I_j, q_i = q_j$ and we may have $I_i = I_j, q_i \neq q_j$). The non-constructive definition below accommodates a wide range of possible constructions. The definition of a *canonical interpolant automaton* in Section 5 is constructive.

Definition 4 (Nested Interpolant Automaton $\mathcal{A}_\mathcal{I}$)**.** *Given a sequence of predicates ("interpolants") $\mathcal{I} = I_0, I_1, \ldots, I_n$, the* nested interpolant automaton

$$\mathcal{A}_\mathcal{I} = (Q_\mathcal{I}, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}}\rangle, Q^{\mathsf{init}}_\mathcal{I}, Q^{\mathsf{fin}}_\mathcal{I})$$

is a finite nested word automaton if we can index its set of states $Q_\mathcal{I}$ with the set of indices of the sequence $\{0, \ldots, n\}$, i.e.,

$$Q_\mathcal{I} = \{q_0, \ldots, q_n\}$$

and thus associate each interpolant I_i with a state q_i, such that the following three conditions hold.

- *The sequence of interpolants is inductive wrt. the triple of transition relations $\langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}}\rangle$, i.e.,*

$$(q_i, \boxed{\texttt{y:=t}}, q_j) \in \delta_{\mathsf{in}} \Rightarrow post_{\mathsf{in}}(I_i, \boxed{\texttt{y:=t}}) \subseteq I_j$$

$$(q_i, \boxed{\phi}, q_j) \in \delta_{\mathsf{in}} \Rightarrow post_{\mathsf{in}}(I_i, \boxed{\phi}) \subseteq I_j$$

$$(q_i, \boxed{\texttt{call p}}, q_j) \in \delta_{\mathsf{ca}} \Rightarrow post_{\mathsf{ca}}(I_i, \boxed{\texttt{call p}}) \subseteq I_j$$

$$(q_i, q_k, \boxed{\texttt{return p}}, q_j) \in \delta_{\mathsf{re}} \Rightarrow post_{\mathsf{re}}(I_i, I_k, \boxed{\texttt{return p}}) \subseteq I_j$$

- *Each interpolant associated with an initial state is the* true *predicate.*

$$q_i \in Q^{\mathsf{init}}_\mathcal{I} \Rightarrow I_i = \top$$

- *Each interpolant associated with a final state is the* false *predicate.*

$$q_i \in Q^{\mathsf{fin}}_\mathcal{I} \Rightarrow I_i = \bot$$

Theorem 4. *A nested interpolant automaton $\mathcal{A}_\mathcal{I}$ recognizes a subset of infeasible nested traces.*

$$\mathcal{L}(\mathcal{A}_\mathcal{I}) \cap \mathcal{L}(\mathcal{A}_\Sigma) = \emptyset$$

Proof. We show by induction on the length of a nested trace $\pi = (st_0, \ldots, st_{n-1}, \rightsquigarrow)$ the following. If $q_0, \ldots, q_n$ is a run of $\mathcal{A}_\mathcal{I}$ for $st_0, \ldots, st_{n-1}$ then $\nu_0, \ldots, \nu_n$ is a run of $\mathcal{A}_\Sigma$ only if $\nu_i \in I_i$ for $i = 0, \ldots, n$.

We proof the induction step for the case where the last position of the sequence is an internal position as follows. Let $q_0, \ldots, q_{n+1}$ be a run of $\mathcal{A}_\mathcal{I}$ for $\pi = (st_0, \ldots, st_n, \rightsquigarrow)$, let $\nu_0, \ldots, \nu_{n+1}$ be a run of $\mathcal{A}_\Sigma$ for π, let position n be an internal position of π. By induction hypothesis $\nu_n \in I_n$. Since $(\nu_n, st_n, \nu_{n+1}) \in \delta_{\mathsf{in}}$, the state ν_{n+1} is in $post_{\mathsf{in}}(I_n, st_n)$. The transition relation of $\mathcal{A}_\mathcal{I}$ contains the triple (q_n, st_n, q_{n+1}). Hence the inclusion $post_{\mathsf{in}}(I_n, st_n) \subseteq I_{n+1}$ is valid. Thus, $\nu_{n+1} \in I_{n+1}$ holds. The cases where n is a call position or a return position can be proven analogously.

Assume $q_0, \ldots, q_n$ is an accepting run of $\mathcal{A}_\mathcal{I}$ for a nested trace π. Therefore I_n, the interpolant associated to q_n, is the *false* predicate $\bot$, which is not satisfied by any valuation. According to the preceding inductive argument there is no run of $\mathcal{A}_\Sigma$ for π, hence π is infeasible. □

Nested Interpolant Automata from Floyd-Hoare Style Proofs. Given a Floyd-Hoare style proof of partial correctness for the program $\mathcal{P}$ in the form of a sequence of inductive invariant assertions $\mathcal{I} = (I_\ell)_\ell$ (the sequence is indexed by the program locations ℓ of $\mathcal{P}$, like in Figure 1), we obtain readily that we can use the control automaton $\mathcal{A}_\mathcal{P}$ as a nested interpolant automaton $\mathcal{A}_\mathcal{I}$. In fact, the conditions in Definition 4 translate exactly the inductiveness of the invariant assertions in the Floyd-Hoare style proof (as presented, e.g., in [4]).

Completeness of Nested Interpolant Automata. We may ask whether the proof rule (1) is still complete if we require that the nested word automaton $\mathcal{A}$ used a proof argument must be the complement of a nested interpolant automaton. By the discussion above, we obtain the relative completeness in the same sense as for the Floyd-Hoare proof rule.

5. Interpolant Automata from Proofs

In this section we will give a constructive proof method for program correctness. In the first step of the method, an inductive sequence of Craig interpolants is generated from an infeasibility proof of a nested error trace (Section 5.3). In the second step, this sequence is used to construct a nested interpolant automaton that recognizes traces infeasible for the same reason (Section 5.4). The two steps are embedded in a counterexample-guided abstraction refinement scheme (Section 5.5).

Before we present the generation of the inductive sequence of Craig interpolants, we give the definition of an inductive sequence of nested interpolants (Section 5.1) and we show how the infeasibility of a nested error trace can be characterized as the satisfiability of a formula (Section 5.2).

5.1 Inductive Sequence of Nested Interpolants

We generalize the concept of a sequence of interpolants [13] to account for the nesting structure. The key idea is that on a return statement the predicate describing the calling context can be combined with the predicate before the return statement that captures the post-condition of the procedure. This is a necessary prerequisite to obtain predicates that only restrict the variables in the current calling context.

Definition 5 (Inductive Sequence of Nested Interpolants)**.** *Given a nested trace*

$$\pi = (st_0 \dots st_{n-1}, \rightsquigarrow)\,,$$

an inductive sequence of nested interpolants is a sequence of predicates $I_0, \dots, I_n$ such that

- $I_0 = \top$,
- $I_n = \bot$,
- *For $i = 0, \dots, n-1$:*
  - $post_{\mathsf{in}}(I_i, st_i) \subseteq I_{i+1}$ *if i is an internal position,*
  - $post_{\mathsf{ca}}(I_i, st_i) \subseteq I_{i+1}$ *if i is a call position,*
  - $post_{\mathsf{re}}(I_i, I_k, st_i) \subseteq I_{i+1}$ *if i is a return position and $k \rightsquigarrow i$.*

Remark. If a nested trace π has an inductive sequence of nested interpolants, then π must be infeasible. In our setting we are only interested in the case were π is an infeasible nested error trace. In Section 5.3 we explain how an inductive sequence of nested interpolants can be obtained from Craig interpolants in that case.

5.2 Infeasibility Proof for Nested Traces

From now on, we assume that the terms in assignment statements and the formulas in assume statements are given in some (first- or higher-order) theory with equality.

For a well-nested trace $(\pi, \rightsquigarrow)$ we construct a formula in this theory, called the single static assignment (SSA) for the trace, which is unsatisfiable if and only if the nested trace is infeasible. If the theory is decidable this allows us to obtain a proof of infeasibility for $(\pi, \rightsquigarrow)$ automatically, by deciding satisfiability of the SSA.

The single static assignment contains a fresh logical variable for each assignment statement in the trace. This allows us to express these statements by equalities. For recursive programs the variables of different calling contexts have to be kept separate. We use the nesting structure to replace each program variable by the last assigned logical variable that belongs to the same procedure.

Definition 6 (SSA)**.** *The single static assignment (SSA) for a well-nested trace $\pi = (st_0, \dots, st_{n-1}, \rightsquigarrow)$ is the sequence of formulas $\varphi_0, \dots, \varphi_{n-1}$ where φ_i is defined as*

- $y^i = \mathsf{rename}_i(t)$ *if* $st_i = \boxed{\texttt{y:=t}}$,
- $\mathsf{rename}_i(\phi)$ *if* $st_i = \boxed{\phi}$,
- $x^i = \mathsf{rename}_i(x_p)$ *if* $st_i = \boxed{\texttt{call p}}$,
- $res_p^i = \mathsf{rename}_i(res)$ *if* $st_i = \boxed{\texttt{return p}}$.

The function rename_i *replaces every variable v by v^j where $j =$* $\mathsf{index}(v, i)$ *is the largest index before i in the same calling context where x is assigned. Formally* $\mathsf{index}(v, 0) = -1$ *and*

$$\mathsf{index}(v, i{+}1) = \begin{cases} & \textit{if } st_i = \boxed{\texttt{y:=t}} \textit{ and } v = y, \\ i & \textit{or } st_i = \boxed{\texttt{return p}} \textit{ and } v = res_p, \\ & \textit{or } i \textit{ is call position}, \\ \mathsf{index}(v, k) & \textit{else if } k \rightsquigarrow i \textit{ (} i \textit{ is return position)}, \\ \mathsf{index}(v, i) & \textit{else (} i \textit{ is internal position)}. \end{cases}$$

The function index is defined recursively and follows the trace from i backwards to the statement where v is assigned to. The assignment can be either explicitly by an assignment or implicitly, e.g., when a variable is used uninitialized in a procedure. If a variable is used without prior initialization it returns the index of the procedures call statement and the index -1 in the case where this procedure is the main procedure of the program. The rules for call, return, and internal positions ensure that the index of the variable $\mathsf{rename}_i(y)$ is in the same procedure as i. To prove the correctness of our interpolation scheme we need the following properties of the renaming function.

Lemma 1. *The function* index *in Definition 6 for a well-nested trace $\pi = (st_0 \dots st_{n-1}, \rightsquigarrow)$ has the following properties:*

1. *Every variable is assigned before the current position, i.e.,* $\mathsf{index}(v, i) < i$*.*
2. *If $k \rightsquigarrow j$, and $k < i \leq j$, then $k \leq \mathsf{index}(v, i)$, i.e., the variable is assigned inside the current procedure. This also holds for an unfinished call, i.e., $k \rightsquigarrow \infty$ and $k < i$.*
3. *If $k \rightsquigarrow j$, and $k < j < i$, i.e., position i is after the position a procedure returns, then either $\mathsf{index}(v, i) \geq j$ or $\mathsf{index}(v, i) < k$, i.e., the variable was not assigned inside that procedure call.*
4. *If $st_i = \boxed{\texttt{y:=t}}$ then for all $j > i$: $\mathsf{index}(y, j) \neq \mathsf{index}(y, i)$, i.e., later statement will never see the value overwritten by statement i.*
5. *If $st_i = \boxed{\texttt{return p}}$ and k is the corresponding call position (i.e. $k \rightsquigarrow i$) then $\mathsf{index}(res_p, j) \neq \mathsf{index}(res_p, k)$ for all $j > i$, i.e., the previous value of the result variable that was overwritten by the return is not visible after the return.*

Proof. 1. This is easily shown by induction over i.

2. Follows by induction and the fact that (a) $\mathsf{index}(v, k+1) = k$ and (b) there can be no nesting $k' \rightsquigarrow j'$ with $k' < k < j' < i < j$ due to proper nesting.
3. Follows by induction. For the induction step recall that there is no nesting $k' \rightsquigarrow j'$ with $k \leq k' \leq j < j'$ due to proper nesting.
4. By induction: For $j = i + 1$, we have $\mathsf{index}(y, i) < i = \mathsf{index}(y, j)$. Now assume the hypothesis for $j > i$ holds. The cases $\mathsf{index}(y, j+1) = j > \mathsf{index}(y, i)$ and $\mathsf{index}(y, j+1) = \mathsf{index}(y, j) \neq \mathsf{index}(y, i)$ are obvious. If $\mathsf{index}(y, j+1) = \mathsf{index}(y, k)$ with $k \rightsquigarrow j$ there are two cases. If $k > i$, then $\mathsf{index}(y, k) \neq \mathsf{index}(y, i)$ by induction hypothesis. Otherwise, $k < i < j$, hence $\mathsf{index}(y, k) < k \leq \mathsf{index}(y, i)$. Since the trace is well-nested, i must be an internal position, so $k = i$ is not possible. This shows that $\mathsf{index}(y, j) \neq \mathsf{index}(y, i)$ for all $j > i$.
5. This is analogous to 4. □

We can check the feasibility of a nested trace by checking the satisfiability of its SSA.

Theorem 5. *Given the SSA $\varphi_0, \dots, \varphi_{n-1}$ for the well-nested trace π. The trace π is feasible if and only if the conjunction $\varphi_0 \wedge \dots \wedge \varphi_{n-1}$ is satisfiable.*

Proof. We obtain a run of the data automaton $\mathcal{A}_\Sigma$ on the trace π from a model $\mathcal{M}$ of $\varphi_0 \wedge \dots \wedge \varphi_{n-1}$ if we set $\nu_i(v)$ to

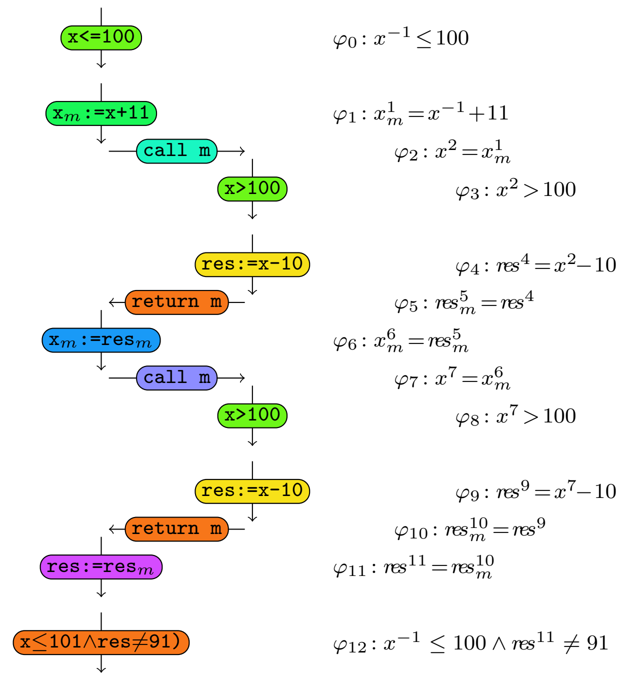

x<=100
$\varphi_0: x^{-1} \leq 100$
x$_m$:=x+11
$\varphi_1: x^1_m = x^{-1}+11$
call m
$\varphi_2: x^2 = x^1_m$
x>100
$\varphi_3: x^2 > 100$
res:=x-10
$\varphi_4: \mathit{res}^4 = x^2 - 10$
return m
$\varphi_5: \mathit{res}^5_m = \mathit{res}^4$
x$_m$:=res$_m$
$\varphi_6: x^6_m = \mathit{res}^5_m$
call m
$\varphi_7: x^7 = x^6_m$
x>100
$\varphi_8: x^7 > 100$
res:=x-10
$\varphi_9: \mathit{res}^9 = x^7 - 10$
return m
$\varphi_{10}: \mathit{res}^{10}_m = \mathit{res}^9$
res:=res$_m$
$\varphi_{11}: \mathit{res}^{11} = \mathit{res}^{10}_m$
x≤101∧res≠91)
$\varphi_{12}: x^{-1} \leq 100 \wedge \mathit{res}^{11} \neq 91$

Figure 6. The nested trace π_4 (a nested error trace of $\mathcal{A}_{\mathcal{P}91}$) and its SSA.

$\mathcal{M}(\mathsf{rename}_i(v))$. Vice versa, we obtain a model from a run $\nu_0, \dots, \nu_n$ if we assign to v^i the value $\nu_{i+1}(v)$. $\square$

Example 4. *Consider the nested trace π_4 in Figure 6 and its SSA. The nested trace is infeasible, since the conjunction of the formulas in the SSA is unsatisfiable. In fact*

$$x^7 = x^6_m = \mathit{res}^5_m = \mathit{res}^4 = x^2 - 10 = x^1_m - 10 = x^{-1} + 1 \leq 101$$

together with $x^7 > 100$ implies $x^7 = 101$. This implies

$$\mathit{res}^{11} = \mathit{res}^{10}_m = \mathit{res}^9 = x^7 - 10 = 91,$$

which contradicts φ_{12}.

5.3 Inductive Sequence of Nested Craig Interpolants

In this section we give a method to generate an inductive sequence of nested interpolants. The method is based on Craig interpolants, which can be computed automatically from unsatisfiability proofs.

Craig Interpolant. Given two formulas ψ^- and ψ^+ whose conjunction is unsatisfiable, a Craig interpolant is a formula ψ such that

- ψ^- implies ψ,
- the conjunction $\psi \wedge \psi^+$ is unsatisfiable, and
- ψ contains only variables that occur in both, ψ^- and ψ^+.

In the remaining section we assume the SSA is given in a theory where each pair of formulas ψ^- and ψ^+ whose conjunction is unsatisfiable has a Craig interpolant. The concept of Craig interpolation can be lifted to a sequence of formulas whose conjunction is unsatisfiable [18] and applied to the SSA of an infeasible trace. For a flat program execution, Craig interpolants derived this way can be used to generate an inductive sequence of interpolants.

If we apply this method to the SSA of a nested trace, a variable of the parent context that is set before calling a procedure and read after the return of the procedure would be allowed to appear in the Craig interpolants for the called procedure. However, this variable is not in the scope of the called procedure and leads to interpolants which are not suitable for a modular analysis. Therefore, we seek Craig interpolants containing only variables that belong to the current calling context.

By moving the instructions of the parent procedure to the end of the sequence as in [14], it is possible to obtain interpolants that are local to the current calling context. We additionally require that the sequence of interpolants is inductive. Therefore, we follow the basic idea of [14] but compute the interpolants from front to back, using the previously computed interpolant as summary that replaces its preceding statements. Instead of conjuncting all preceding statements of the parent procedure we use the previously computed interpolant directly preceding the call statement. This algorithm gives us an inductive sequence of nested interpolants.

Definition 7 (Inductive Sequence of Nested Craig Interpolants)**.** *Given an SSA $\varphi_0 \dots, \varphi_{n-1}$ and a nesting relation $\rightsquigarrow$, an inductive sequence of nested Craig interpolants is a sequence $\psi_0 \dots \psi_n$, where*

- *ψ_0 is $\top$.*
- *If $\psi_0, \dots, \psi_i$ ($i \in \{0, \dots, n-1\}$) are given, we obtain ψ_{i+1} as follows. We define*

$$\psi^+_{i+1} = \bigwedge_{j=i+1}^{n-1} \varphi_j \wedge \bigwedge_{\substack{k \rightsquigarrow j,\\ k<i+1<j\neq\infty}} (\psi_k \wedge \varphi_k),$$

which is the conjunction of the statements following position $i+1$ and the pre-conditions of the calls which are pending at position $i+1$, and

$$\psi^-_{i+1} = \begin{cases} \psi_i \wedge \varphi_i & \text{if } i \text{ is internal position,} \\ \psi_i \wedge \varphi_i & \text{if } i \text{ is call position, } i \rightsquigarrow \infty, \\ \top & \text{if } i \text{ is call position, } i \not\rightsquigarrow \infty, \\ \psi_i \wedge \varphi_i \wedge \psi_k \wedge \varphi_k & \text{if } i \text{ is return position, } k \rightsquigarrow i. \end{cases}$$

Then we obtain ψ_{i+1} as a Craig interpolant of ψ^-_{i+1} and ψ^+_{i+1}.

The following lemma is needed for the correctness of the interpolation scheme.

Lemma 2. *Given an infeasible well-nested trace*

$$\pi = (\mathit{st}_0 \dots \mathit{st}_{n-1}, \rightsquigarrow)$$

and its SSA $\varphi_0, \dots, \varphi_{n-1}$.

1. *For $i = 1, \dots, n$, the conjunction $\psi^-_i \wedge \psi^+_i$ in Definition 7 is unsatisfiable, and therefore the Craig interpolant exists.*
2. *For $i = 0, \dots, n$, the interpolant ψ_i contains only the variables $\mathsf{rename}_i(v)$ for program variables v.*
3. *For $i = 0, \dots, n-1$,*
   $\psi_i \wedge \varphi_i \Rightarrow \psi_{i+1}$ if i is an internal or a call position and
   $\psi_i \wedge \varphi_i \wedge \psi_k \wedge \varphi_k \Rightarrow \psi_{i+1}$ if i is a return positions and $k \rightsquigarrow i$.
4. *$\psi_n = \bot$.*

Proof. 1. We show this by induction over i. For $i = 1$ the conjunction $\psi^-_1 \wedge \psi^+_1$ is equal to $\varphi_0 \wedge \dots \wedge \varphi_{n-1}$, which is unsatisfiable by Theorem 5. For the induction step note that the conjunction $\psi^-_{i+1} \wedge \psi^+_{i+1}$ has exactly the same conjuncts as $\psi_i \wedge \psi^+_i$. Since ψ_i was obtained as the Craig interpolant of ψ^-_i and ψ^+_i (which exists by induction hypothesis), the formula $\psi_i \wedge \psi^+_i$ is unsatisfiable.

2. This is also shown by induction. For $i = 0$ this is obvious as $\psi_0 = \top$. Now assume that ψ_k contains only the variables

$\mathsf{rename}_k(v)$ for $k \le i$. All variables appearing in ψ_{i+1} must also appear in ψ^-_{i+1} and ψ^+_{i+1}.
If $\mathit{st}_i = \boxed{\texttt{y:=t}}$ then ψ^-_{i+1} contains only $\mathsf{rename}_i(v)$ and y^i. By definition $\mathsf{rename}_{i+1}(v) = \mathsf{rename}_i(v)$ except for $v \equiv y$. It remains to show that the variable $\mathsf{rename}_i(y)$ does not appear in ψ^+_{i+1}. The formula $\bigwedge_{j=i+1}^{n-1} \varphi_i$ contains only $\mathsf{rename}_j(y)$ for $j > i$. Lemma 1.4 ensures $\mathsf{rename}_j(y) \ne \mathsf{rename}_i(y)$. The second conjunction of ψ^+_{i+1} contains the variables $\mathsf{rename}_k(v)$, and x^k for procedure calls $k \rightsquigarrow j$ with $k < i+1 < j < \infty$. Lemma 1.2 ensures $\mathsf{index}(v,k) < k \le \mathsf{index}(y,i)$, hence $\mathsf{rename}_k(v) \ne \mathsf{rename}_i(y)$. Since the input variable x is never written, $x \not\equiv y$.
If $\mathit{st}_i = \boxed{\phi}$, then by the induction hypothesis and the definition of φ_i, $\psi^-_{i+1} = \psi_i \wedge \varphi_i$ contains only the variables $\mathsf{rename}_i(v)$. By definition, $\mathsf{rename}_{i+1}(v) = \mathsf{rename}_i(v)$. Thus, ψ_{i+1} contains only the allowed variables.
If $\mathit{st}_i = \boxed{\texttt{call p}}$ is an unfinished call statement, i.e., $i \rightsquigarrow \infty$, then $\psi^-_{i+1} = \psi_i \wedge \varphi_i$ contains the variables $\mathsf{rename}_i(v)$ and x^i. The formula ψ^+_i contains only variables $\mathsf{rename}_j(v)$ with $j > i$ (due to proper nesting there is no call $k \rightsquigarrow j$ with $k < i < j < \infty$). By Lemma 1.2 $\mathsf{index}(v,i) < i \le \mathsf{index}(v,j)$, hence $\mathsf{rename}_i(v) \ne \mathsf{rename}_j(v)$. The variable $x^i = \mathsf{rename}_{i+1}(x)$ is allowed to appear in ψ_{i+1}.
If i is a call position with $i \not\rightsquigarrow \infty$, then $\psi^-_{i+1} = \top$ hence ψ_{i+1} does not contain any variable.
If $\mathit{st}_i = \boxed{\texttt{return p}}$ and $k \rightsquigarrow i$ is the corresponding call position, then $\psi^-_{i+1} = \psi_i \wedge \varphi_i \wedge \psi_k \wedge \varphi_k$ contains the variables $\mathsf{rename}_i(v)$, $\mathsf{rename}_k(v)$, x^k, and res^i_p. It is $\mathsf{rename}_{i+1}(v) = \mathsf{rename}_k(v)$ except for $v \equiv \mathit{res}_p$ and $\mathsf{rename}_{i+1}(\mathit{res}_p) = \mathit{res}^i_p$. It remains to show that $\mathsf{rename}_i(v)$, $\mathsf{rename}_k(\mathit{res}_p)$, and x^k do not appear in ψ^+_{i+1}. The formula ψ^+_{i+1} contains $\mathsf{rename}_j(v)$ for $j > i$, $\mathsf{rename}_{k'}(v)$, and $x^{k'}$ for $k' \rightsquigarrow j'$ with $k' < i < j'$. By proper nesting $k' < k$ holds. For $j > i$ we have $\mathsf{index}(\mathit{res}_p, j) \ne \mathsf{index}(\mathit{res}_p, k)$ by Lemma 1.5 and for $k' < k < j'$ Lemma 1.2 gives $\mathsf{index}(\mathit{res}_p, k) \ge k' > \mathsf{index}(\mathit{res}_p, k')$. Hence $\mathsf{rename}_k(\mathit{res}_p)$ does not appear in ψ^+_{i+1}. The indices of the variables x^k and $\mathsf{rename}_i(v)$ are all between k inclusive and i exclusive. For $j > i$ Lemma 1.3 gives $\mathsf{index}(v,j) \ge i$ or $\mathsf{index}(v,j) < k$ and for $k' < k$ the index of $\mathsf{rename}_{k'}(v)$ and $x^{k'}$ is smaller than k. Therefore, the variables x^k and $\mathsf{rename}_i(v)$ do not appear in ψ^+_{i+1}.

3. These implications follow directly from the property $\psi^-_{i+1} \Rightarrow \psi_{i+1}$ of Craig interpolants.
4. Finally, $\psi_n = \bot$ follows from $\psi^+_n = \top$, because $\psi_n \wedge \psi^+_n$ is unsatisfiable. □

We extend the renaming function to valuations

$$\mathsf{rename}_i(\nu) = \{\mathsf{rename}_i(v) \mapsto \nu(v) \mid v \text{ is variable}\}.$$

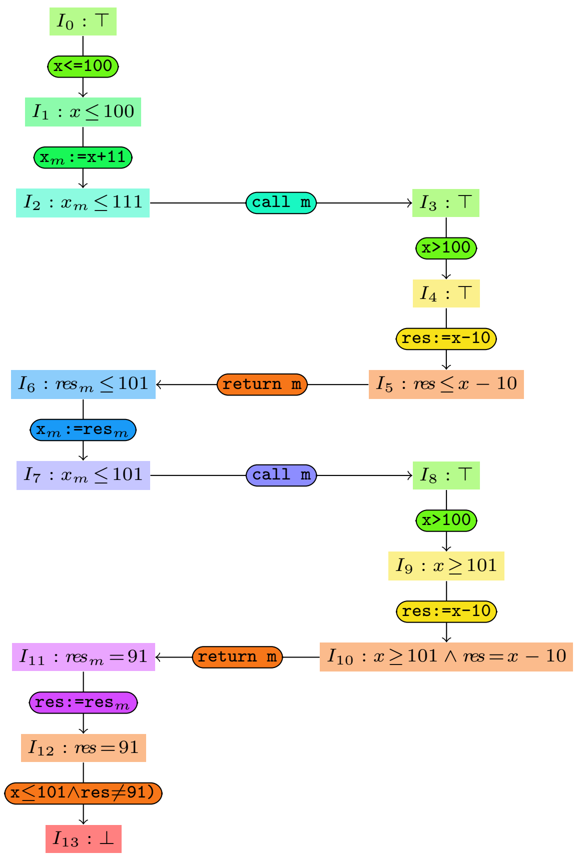
Since the Craig interpolant ψ_i contains only $\mathsf{rename}_i(v)$ as variables, we can associate it with the set of valuations ν such that $\mathsf{rename}_i(\nu) \models \psi_i$.

Theorem 6. *Given a well-nested trace π, its SSA $\varphi_0, \dots, \varphi_{n-1}$, and its inductive sequence of nested Craig interpolants $\psi_0, \dots, \psi_n$, the sequence of predicates $I_0, \dots, I_n$ with*

$$I_i = \{\nu \mid \mathsf{rename}_i(\nu) \models \psi_i\} \quad \textit{for } i = 0, \dots, n$$

is an inductive sequence of nested interpolants of π.

Proof. For brevity, we write $\nu_i \models \psi_i$ for $\mathsf{rename}_i(\nu_i) \models \psi_i$, which means $\nu_i \in I_i$ Similarly, for ν_i, ν_j we write $\nu_i, \nu_j \models \varphi$ as short-



$I_0 : \top$
x<=100
$I_1 : x \le 100$
$\texttt{x}_m\texttt{:=x+11}$
$I_2 : x_m \le 111$
call m
$I_3 : \top$
x>100
$I_4 : \top$
res:=x-10
$I_5 : \mathit{res} \le x - 10$
return m
$I_6 : \mathit{res}_m \le 101$
$\texttt{x}_m\texttt{:=res}_m$
$I_7 : x_m \le 101$
call m
$I_8 : \top$
x>100
$I_9 : x \ge 101$
res:=x-10
$I_{10} : x \ge 101 \wedge \mathit{res} = x - 10$
return m
$I_{11} : \mathit{res}_m = 91$
$\texttt{res:=res}_m$
$I_{12} : \mathit{res} = 91$
$\texttt{x}\le\texttt{101}\wedge\texttt{res}\ne\texttt{91)}$
$I_{13} : \bot$

Figure 7. Inductive sequence of nested interpolants $\mathcal{I} = (I_0, \dots, I_{13})$, obtained from the Craig interpolants of the SSA in Figure 6. For better legibility the sequences are depicted in an interleaved arrangement.

hand for

$$\mathsf{rename}_i(\nu_i) \cup \mathsf{rename}_j(\nu_j) \models \varphi\,.$$

This is well-defined if $\nu_i(v) = \nu_j(v)$ whenever $\mathsf{rename}_i(v) = \mathsf{rename}_j(v)$.

- $\mathit{st}_i = \boxed{\texttt{y:=t}}$: Given $\nu_{i+1} \in \mathit{post}_{\mathsf{in}}(I_i, \mathit{st}_i)$, by definition there is $\nu_i \in I_i$ (hence $\nu_i \models \psi_i$), such that $\nu_{i+1} = \nu_i \oplus \{y \mapsto \nu_i(t)\}$. Since φ_i is $y^i = \mathsf{rename}_i(t)$, we get $\nu_i, \nu_{i+1} \models \varphi_i$. From the previous lemma we have $\psi_i \wedge \varphi_i \Rightarrow \psi_{i+1}$, hence $\nu_i, \nu_{i+1} \models \psi_{i+1}$. The latter formula contains only variables $\mathsf{rename}_{i+1}(v)$, hence $\nu_{i+1} \in I_{i+1}$. This shows $\mathit{post}_{\mathsf{in}}(I_i, \mathit{st}_i) \subseteq I_{i+1}$.
- $\mathit{st}_i = \boxed{\phi}$: Let $\nu_{i+1} \in \mathit{post}_{\mathsf{in}}(I_i, \mathit{st}_i)$, hence $\nu_{i+1} \in I_i$ and $\nu_{i+1} \models \phi$. Since $\mathsf{rename}_i(v) = \mathsf{rename}_{i+1}(v)$ for all variables v and φ_i is $\mathsf{rename}_i(\phi)$, this implies $\nu_{i+1} \models \psi_i \wedge \varphi_i$ and with the previous lemma $\nu_{i+1} \models \psi_{i+1}$. Hence, $\nu_{i+1} \in I_{i+1}$.
- $\mathit{st}_i = \boxed{\texttt{call p}}$: Let $\nu_{i+1} \in \mathit{post}_{\mathsf{ca}}(I_i, \mathit{st}_i)$. By definition there is $\nu_i \in I_i$ and $\nu_{i+1}(x) = \nu_i(x_p)$. By definition of φ_i, this implies $\nu_i, \nu_{i+1} \models \varphi_i$. With $\nu_i \models \psi_i$ and $\psi_i \wedge \varphi_i \Rightarrow \psi_{i+1}$ this implies $\nu_i, \nu_{i+1} \models \psi_{i+1}$. Thus, $\nu_{i+1} \in \psi_{i+1}$.
- $\mathit{st}_i = \boxed{\texttt{return p}}$: Then i is a return position and $k \rightsquigarrow i$. Let $\nu_{i+1} \in \mathit{post}_{\mathsf{re}}(I_i, I_k, \mathit{st}_i)$. Then there is $\nu_i \in I_i$, $\nu_k \in I_k$ with $\nu_i(x) = \nu_k(x_p)$ and $\nu_{i+1} = \nu_k \oplus \{\mathit{res}_p = \nu_i(\mathit{res})\}$. Since φ_i is $\mathit{res}^i_p = \mathsf{rename}_i(\mathit{res})$, this implies $\nu_i, \nu_{i+1} \models \varphi_i$. Furthermore $\nu_i \models \psi_i$ and $\nu_k \models \psi_k$ and $\nu_k, \nu_i \models \varphi_k$, since φ_k is $x^k = \mathsf{rename}_k(x)$ and $\mathsf{rename}_i(x) = x^k$. With the previous lemma it follows $\nu_k, \nu_i, \nu_{i+1} \models \psi_{i+1}$. Hence, $\nu_{i+1} \in I_{i+1}$. □

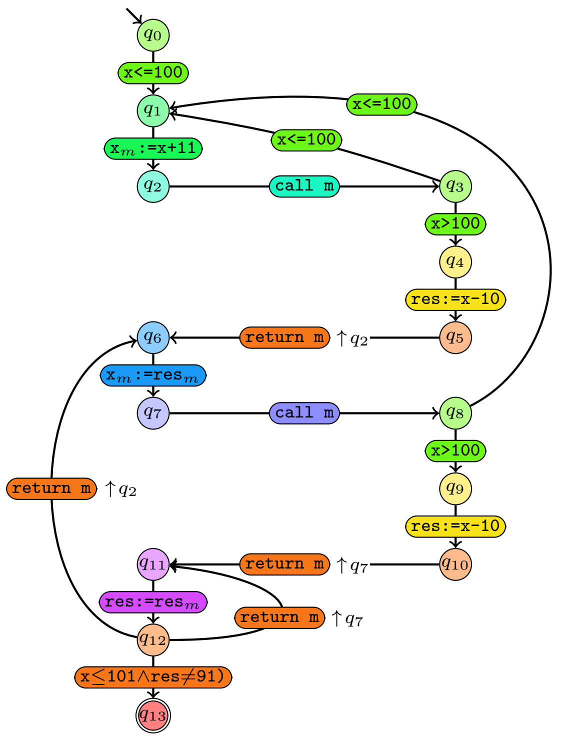


Figure 8. Canonical interpolant automaton $\mathcal{A}_{\mathcal{I}}^{\pi 4}$ for the inductive sequence of nested interpolants depicted in Figure 7. For better legibility we omitted the edges $(q_8,$ x>100$, q_4)$, $(q_9,$ res:=x-10$, q_5)$, $(q_{10}, q_2,$ return m$, q_6)$, which are not needed to prove correctness of $\mathcal{P}^{91}$.

Example 5. *Figure 7 shows the sequence of nested interpolants for the nested trace π depicted in Figure 6. The interpolants were computed as Craig interpolants.*

5.4 Canonical Interpolant Automaton

We use an inductive sequence of nested interpolants for a nested error trace π to construct a nested interpolant automaton. The following construction accepts π and other traces that are infeasible for the same reason as π.

Definition 8 (Canonical Interpolant Automaton $\mathcal{A}_{\mathcal{I}}^{\pi}$). *Given an inductive sequence of nested interpolants $\mathcal{I} = I_0, I_1, \dots, I_n$ of an infeasible error trace $\pi = (st_0 \dots st_{n-1}, \rightsquigarrow)$ along the sequence of locations $\ell_0, \dots, \ell_n$ (the run of the program automaton), we introduce pairwise different states $q_0, \dots, q_n$ and define the* canonical interpolant automaton *$\mathcal{A}_{\mathcal{I}}^{\pi}$ for π and $\mathcal{I}$ as follows.*

$$\mathcal{A}_{\mathcal{I}}^{\pi} = \langle Q, \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle, Q^{\mathsf{init}}, Q^{\mathsf{fin}} \rangle$$

- $Q_{\mathcal{I}} = \{q_0, \dots, q_n\}$

- $\delta_{\mathsf{in}} = \{(q_i, st_j, q_{j+1}) \mid j \text{ is an internal position}, \; i, j = 0, \dots, n-1, \; j \leq i, \; \ell_i = \ell_j, \; post_{\mathsf{in}}(I_i, st_j) \subseteq I_{j+1}\}$
  $\delta_{\mathsf{ca}} = \{(q_i, st_j, q_{j+1}) \mid j \text{ is a call position}, \; i, j = 0, \dots, n-1, \; j \leq i, \; \ell_i = \ell_j, \; post_{\mathsf{ca}}(I_i, st_j) \subseteq I_{j+1}\}$
  $\delta_{\mathsf{re}} = \{(q_i, q_k, st_j, q_{j+1}) \mid j \text{ is a return position}, k \rightsquigarrow j \; i, j = 0, \dots, n-1, \; j \leq i, \; \ell_i = \ell_j, \; post_{\mathsf{re}}(I_i, I_k, st_j) \subseteq I_{j+1}\}$
- $Q_{\mathcal{I}}^{\mathsf{init}} = \{q_0\}$
- $Q_{\mathcal{I}}^{\mathsf{fin}} = \{q_n\}$

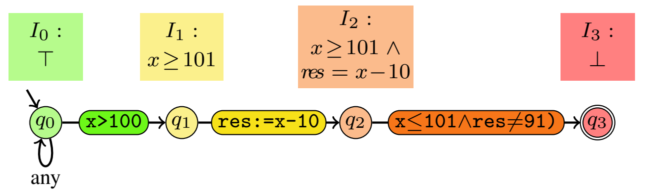


Figure 9. Interpolant automaton $\mathcal{A}_{\mathcal{I}}^{\pi 3}$ that accepts all well-nested traces with the suffix x>100. res:=x-10. x≤101∧res≠91). In combination with the interpolant automata $\mathcal{A}_{\mathcal{I}}^{\pi 4}$ this automaton is sufficient to prove correctness of the program $\mathcal{P}^{91}$ with our proof rule.

The canonical interpolant automaton $\mathcal{A}_{\mathcal{I}}^{\pi}$ accepts the nested error trace π. This follows from the definition of an inductive sequence of nested interpolants. In general $\mathcal{A}_{\mathcal{I}}^{\pi}$ recognizes an infinite set of traces. In a sense, $\mathcal{A}_{\mathcal{I}}^{\pi}$ accepts exactly the traces that are infeasible for the same reason as π. More precisely, in order to prove the infeasibility of a trace accepted by $\mathcal{A}_{\mathcal{I}}^{\pi}$, we can use the same sequence of nested interpolants (up to repetition of subsequences) as in the proof of infeasibility of π.

For $j = i$ the conditions in δ_{in}, δ_{ca}, and δ_{re} hold by Definition 5. Thus, after having generated the inductive sequence of nested interpolants $\mathcal{I}$ (for the proof of the infeasibility of the trace π), one needs additional theorem prover calls only for each inclusion $post_{\mathsf{in}}(I_i, st_{j+1}) \subseteq I_{i+1}$, $post_{\mathsf{ca}}(I_i, st_{j+1}) \subseteq I_{i+1}$, resp. $post_{\mathsf{re}}(I_i, I_k, st_{j+1}) \subseteq I_{i+1}$ where $j < i$ and $\ell_i = \ell_j$. Thus, the number of additional theorem prover calls is bounded by the number of repeated locations in the sequence of locations along the error trace π.

Example 6. *Figure 8 shows the canonical interpolant automaton for the trace π in Figure 6 and its inductive sequence of nested interpolants in Figure 7.*

The interpolant automaton is not sufficient to prove the correctness of the program using proof rule (1). The automaton $\mathcal{A}_{\mathcal{I}}^{\pi 4}$ does not accept the nested error trace π_3 from Figure 4. Hence, for the complement automaton $\overline{\mathcal{A}_{\mathcal{I}}^{\pi 4}}$ the precondition of proof rule (1)

$$\mathcal{L}(\overline{\mathcal{A}_{\mathcal{I}}^{\pi 4}}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset$$

does not hold.

Figure 9 shows a second interpolant automaton $\mathcal{A}_{\mathcal{I}}^{\pi 3}$ that accepts π_3 and infeasible nested traces similar to π_3. The product automaton of the complemented interpolant automata $\overline{\mathcal{A}_{\mathcal{I}}^{\pi 4}} \cap \overline{\mathcal{A}_{\mathcal{I}}^{\pi 3}}$ still recognizes a superset of feasible traces

$$\mathcal{L}(\overline{\mathcal{A}_{\mathcal{I}}^{\pi 4}} \cap \overline{\mathcal{A}_{\mathcal{I}}^{\pi 3}}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma}).$$

It can be mechanically checked that

$$\mathcal{L}(\overline{\mathcal{A}_{\mathcal{I}}^{\pi}} \cap \overline{\mathcal{A}_{\mathcal{I}}^{\pi 3}}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}^{91}}) = \emptyset$$

which proves the correctness of $\mathcal{P}^{91}$ with proof rule (1).

5.5 Counterexample-Guided Abstraction Refinement

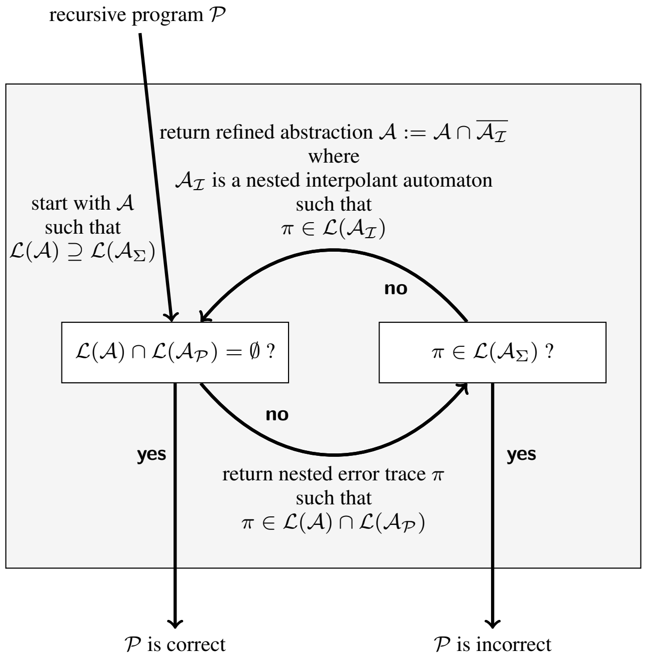
The example of the last section motivates the iterated refinement scheme depicted in Figure 10, similar to the classical check-analyze-refine loop [5, 10].

We start with a coarse abstraction $\mathcal{A}$ of the data automaton $\mathcal{A}_{\Sigma}$, e.g., the automaton that accepts any nested trace. If the abstraction is not a proof for correctness, it accepts a nested error trace, say π. We check whether π is infeasible. If this is the case, we obtain a nested interpolant automaton $\mathcal{A}_{\mathcal{I}}$ that accepts π. This automaton can be the canonical interpolant automaton $\mathcal{A}_{\mathcal{I}}^{\pi}$. We refine our abstraction by

$$\mathcal{A} := \mathcal{A} \cap \overline{\mathcal{A}_{\mathcal{I}}}$$

and repeat the loop with the new abstraction.

If we construct a deterministic interpolant automaton $\mathcal{A}_{\mathcal{I}}$, complementing the automaton boils down to inverting the set of final states. The intersection of the program automaton and the interpolant automaton can be computed by unfolding the program automaton, annotating the nodes with the interpolants of the interpolant automaton and merging nodes with the location and same interpolant. This is essentially an extension of the algorithm from [19] to recursive programs.



recursive program $\mathcal{P}$
return refined abstraction $\mathcal{A} := \mathcal{A} \cap \overline{\mathcal{A}_{\mathcal{I}}}$ where $\mathcal{A}_{\mathcal{I}}$ is a nested interpolant automaton such that $\pi \in \mathcal{L}(\mathcal{A}_{\mathcal{I}})$
start with $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma})$
no
$\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset$?
$\pi \in \mathcal{L}(\mathcal{A}_{\Sigma})$?
no
yes
yes
return nested error trace π such that $\pi \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}})$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

Figure 10. Counterexample-guided abstraction refinement scheme that exploits proof rule (1). If $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{A}_{\Sigma})$ and $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{P}}) = \emptyset$ then program $\mathcal{P}$ is correct.

6. Conclusion

In this paper, we have explored the potential of the theory of nested words as a foundation for correctness proofs for the general class of recursive procedures. Our conceptual contribution is a simple framework that allows us to shine a new light on classical concepts such as Floyd/Hoare proofs and predicate abstraction for recursive programs. Our technical contribution is to give, to our knowledge for the first time, a principled method that constructs an abstract proof for recursive programs from interpolants (avoiding the construction of the abstract transformer). We have used nested words to formalize the concept of inductive sequences of interpolants for traces of recursive programs. The construction of nested word automata from inductive sequences interpolants is interesting because it avoids the construction of the abstract transformer. As pointed out in [19], the interpolation-based proof method can be made to scale if one exploits the modular structure of procedural programs (i.e., using procedure summaries). Our work provides the foundation to explore different realizations of this approach.

Acknowledgments

We thank Neil Jones and Domagoj Babić for comments on the presentation of this paper. This work was partly supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

[1] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS '04*, pages 467–481. Springer, 2004.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT '06*, pages 1–13. Springer, 2006.

[3] R. Alur and P. Madhusudan. Adding nesting structure to words. *JACM*, 56(3), 2009.

[4] K.-R. Apt, F. de Boer, and E.-R. Olderog. *Verification of sequential and concurrent programs.* Third, extended edition. Springer, 2009.

[5] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3. ACM, 2002.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, pages 364–387. Springer, 2005.

[7] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08*, pages 3–14. ACM, 2008.

[8] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In *FSEN '07*, pages 17–32. Springer, 2007.

[9] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00*, pages 154–169. Springer, 2000.

[11] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. *JSAT*, 5:27–56, June 2008. Special Issue on Constraints to Formal Verification.

[12] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS '08*, pages 443–458. Springer, 2008.

[13] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS '09*, pages 69–85. Springer, 2009.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL '04*, pages 232–244. ACM, 2004.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, 2002.

[16] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *ICCD '05*, pages 297–308. IEEE Computer Society, 2005.

[17] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV '05*, pages 39–51. Springer, 2005.

[18] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS '06*, pages 459–473. Springer, 2006.

[19] K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136. Springer, 2006.

[20] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL '07*, pages 245–259. Springer, 2007.

[21] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61. ACM, 1995.