Temporal Reasoning for Procedural Programs

Rajeev Alur[1] and Swarat Chaudhuri[2]

[1] University of Pennsylvania, USA
[2] Pennsylvania State University, USA

Abstract. While temporal verification of programs is a topic with a long history, its traditional basis—semantics based on word languages—is ill-suited for modular reasoning about procedural programs. We address this issue by defining the semantics of procedural (potentially recursive) programs using *languages of nested words* and developing a framework for temporal reasoning around it. This generalization has two benefits. First, this style of reasoning naturally unifies Manna-Pnueli-style temporal reasoning with Hoare-style reasoning about structured programs. Second, it allows verification of "non-regular" properties of specific procedural contexts—e.g., "If a lock is acquired in a context, then it is released in the same context." We present proof rules for a variety of properties such as *local safety*, *local response*, and *staircase reactivity*; our rules are sufficient to prove all temporal properties over nested words. We show that our rules are sound and relatively complete.

1 Introduction

A prominent approach to program verification relies on identifying pre and post-conditions for every block. For example, the Hoare triple $\{\varphi\}P\{\psi\}$ for partial correctness means that if we execute the program P starting from a state satisfying the state predicate φ, then if the program terminates, the final state satisfies ψ [12, 5, 8]. The corresponding proof system contains a rule for each of the syntactic constructs for building complex programs, allowing modular proofs of structured programs. The last few years have seen renewed interest in such proofs, largely due to the coming-of-age of powerful decision procedures.

While Hoare-style reasoning can establish functional correctness of programs, it is not well-suited for reasoning about reactive programs. The most widely accepted formalism for verification of reactive programs is temporal logic [17]. In temporal reasoning, the semantics of a program P is defined to be a set of executions, where each execution is a sequence of program states; the specification is a formula φ of linear temporal logic (LTL); and P satisfies φ if all its executions are satisfying models of φ. Manna-Pnueli-style proof systems for temporal logics show how to establish temporal properties of programs by reasoning about state formulas [15, 16]. A limitation of these rules, however, is that they do not exploit the modularity offered by the procedural structure of programs. Also, the temporal properties that they prove cannot refer to specific procedural contexts. For example, the property "If a lock is acquired in a procedural context, then it

is released before the context ends," which refers to the non-regular nesting of procedural contexts, is inexpressible in temporal logic.

There has been, of late, a resurgence of interest in program verification due to the success of model checking tools like SLAM [7]. In most of these settings, even though the analyzed program is sequential, the requirements are temporal (e.g., "Lock A must be acquired *after* lock B"); thus, temporal reasoning is needed. Yet, any verification method that does not exploit the modularity afforded by procedures will not scale to large programs. As a result, a form of *procedure-modular* temporal reasoning seems important to develop. Also, as properties of specific procedural contexts arise naturally in procedural programs, it seems natural to ask for proofs for these. This paper offers a framework for temporal reasoning that satisfies both these criteria.

Here, the execution of a program is modeled as a *nested word* [3, 4]. Nested words are a model of data with both a linear ordering and a hierarchically nested matching of items. In nested-word modeling of program executions, we augment the linear sequencing of program states with *markup tags* matching procedure calls with returns. The benefits of this modeling have already been shown for software model checking: when all variables are boolean, viewing the program as a finite-state nested-word-automaton generating a regular language of nested words allows model checking of non-regular temporal properties [2, 1, 6].

In this paper, we first define a simple procedural language, then define its intensional semantics using nested words. Here, each state has information only about the variables currently in scope, and the procedure stack is not made explicit. Now we use it to develop a framework of modular reasoning for procedural programs. State formulas here can refer to the values of variables in scope as well as to their values when the procedure was invoked. We use them to capture *local invariants* (properties that hold at each reachable state of a procedure) and *summaries* (properties that hold when the procedure returns). The classical notion of inductive invariants is now extended to local invariants. Establishing such invariants requires mutually inductive reasoning using summaries—e.g., to establish a local invariant of a procedure `p` that calls a procedure `q`, we use a summary of `q`, establishing which may require the use of a summary of `p`.

Based on these ideas, we develop proof rules for several safety and liveness properties of procedural programs. In a nested word, there are many notions of paths such as *global*, *local*, and *staircase* [2, 1, 13]—temporal logics for nested words contain modalities such as "always" and "eventually" parameterized by the path type. This makes these logics more expressive than LTL—e.g., we can now express *local safety properties* such as "At all points in the top-level procedural context, φ holds" and *local liveness properties* such as "φ holds eventually in the top-level context."

We show that the classical rules proving safety and liveness using inductive invariants and ranking functions can be generalized to these properties. For example, to prove the local safety property above, we use a local invariant for the top-level procedure `p` that implies φ. Proving local liveness requires us to combine reasoning using local invariants and summaries with ranking-function-

based techniques. Along with known expressiveness results for nested words [13, 6], they ensure that we have a proof system for all temporal logic properties of nested words.

We address soundness and completeness of our proof rules. For example, for *local safety*, we show that our rule is sound; that it is complete provided the set of locally reachable states is definable within the underlying assertion language for writing state properties; and that this set is definable provided the assertion language is first-order and can specify a tree data structure. This establishes *relative completeness* of this rule in the style of Manna and Pnueli [14]. Similar results hold for liveness and properties on global and staircase paths.

The paper is organized as follows. Section 2 recapitulates nested words. Sec. 3 fixes a procedural language, and Section 4 defines local invariants and summaries. Section 5, our main technical section, uses these in temporal verification.

Related Work. Hoare-style assertional reasoning [12, 5] for sequential programs is inherently procedure-modular; local invariants and summaries also show up in this setting [8]. Analysis using summaries is key to interprocedural program analysis [21, 19, 20, 10] and software model checking [7, 11]. The standard references for temporal logic are [15, 16]; see [14] for completeness proofs. The original reference on nested words is a paper by Alur and Madhusudan [4]. There have been many papers on nested words and associated logics recently, but these focus on model checking (of pushdown models) and expressiveness [13, 2, 1, 6].

The paper most relevant to this work is by Podelski et al [18]; it uses summaries to compositionally verify termination and liveness of recursive programs (a mechanization of termination of recursive programs appears in [9]). In contrast, this paper uses a nested word semantics of programs, and handles all properties specifiable in temporal logics over nested words, including those explicitly referring to procedural contexts.

2 Nested words

Let Σ be an alphabet and $\texttt{<}, \texttt{>} \notin \Sigma$ be two symbols respectively known as the *call and return tags*. For a word w and $i \in \mathbb{N}$, let $w(i)$ denote the symbol at the i-th position of w; and for $i, j \in \mathbb{N}$ and $j < i$, let w_{ji} denote the word $w_j w_{j+1} \dots w_i$. Let a word w_{ji} as above be *matched* if it is of the form $w ::= ww \mid \sigma \mid \texttt{<}w\texttt{>}$, where σ ranges over Σ. A *nested word* over Σ is now defined to be a finite or infinite word w over $(\Sigma \cup \{\texttt{<}, \texttt{>}\})$ such that for each i with $w(i) = \texttt{>}$, there is a $j < i$ such that $w(j) = \texttt{<}$ and w_{ji} is matched.

A position i in w (positions are numbered $0, 1, \dots$) is a *call* if $w(i+1) = \texttt{<}$, and a *return* if $w(i-1) = \texttt{>}$. If i is a call, j is a return, and w_{ij} is matched, then j is the *matching return* of i. Calls without matching returns are *pending*. For example, consider a nested word $w' = s_0 s_1 \texttt{<} s_3 \texttt{<} s_5 \texttt{<} s_7 \texttt{>} s_9 \texttt{>} s_{11}$. Here, position 1 is a call (as $w(2) = \texttt{<}$), 9 is a return, 1 is a pending call, and 9 is the matching return of 5. A *language of nested words* is a set L of nested words.

Intuitively, we use nested words to model executions of procedural programs, and languages of nested words to define a program's intensional semantics. We

interpret Σ as the set of program states, and the call and return tags as respectively marking the beginning and end of procedural contexts. Call and return positions respectively model the points right before and after control enters/exits a context, while a pending call is a call that does not terminate.

Notably, nested words can also be defined as a logical structure that enriches a word with a matching relation [6, 1]. The present definition may be seen as defining a *linear encoding* of such structures.

Local, global, and staircase paths. The markup provided by the call/return tags in a nested word allows us to distinguish between the parts of the word corresponding to different procedural contexts. These "parts" are naturally viewed as subsequences. Of them, three are of particular interest.

The *global path* in w is the word obtained by removing all call and return tags from w. The *local path* in w is the word w' obtained by erasing from w: (1) every sub-word w_{jk} such that $w(j) = \texttt{<}$, $w(k) = \texttt{>}$, and w_{jk} is matched; and (2) the suffix of w starting at the position $(i+1)$, for the least i such that $w(i)$ is a pending call. For example, the local path in our example nested word w' is $s_0 s_1$.

The *staircase path* in w is the word w' obtained by first erasing from w every sub-word w_{jk} such that $w(j) = \texttt{<}$, $w(k) = \texttt{>}$, and w_{jk} is matched, and then erasing all call tags from the word that results. For example, the staircase path in our example nested word w' is $s_0 s_1 s_3 s_{11}$.

Intuitively, if w models a program execution, then the values of its global variables flow along its global path. The local path of captures the flow of local data in the "top-level" procedural context. If a local path reaches a call that eventually returns, it "jumps" to its matching return; if it reaches a pending call, it terminates. Staircase paths also skip across terminating procedure calls. Unlike local paths, they continue into the new context on seeing a pending call.

3 A simple procedural language

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & [\texttt{global}\ \mathit{Gdec}]\ \mathit{Pdec} \\
\mathit{Gdec} & ::= & \texttt{x} \mid \mathit{Gdec}\ ,\ \mathit{Gdec} \\
\mathit{Ldec} & ::= & \texttt{x} = \mathit{AConst} \mid \mathit{Ldec}\ ,\ \mathit{Ldec} \\
\mathit{Pdec} & ::= & \texttt{proc p}() = \mathit{Pbody} \mid \mathit{Pdec}\ \mathit{Pdec} \\
\mathit{Pbody} & ::= & [\texttt{local}\ \mathit{Ldec}]\ \ \mathit{Com} \\
\mathit{Com} & ::= & l\colon \texttt{skip} \mid l\colon \texttt{x} := \mathit{Aexp} \mid l\colon \texttt{p}() \\
 & & \mid \mathit{Com}; \mathit{Com} \mid l\colon \texttt{while}\ \ \mathit{Bexp}\ \texttt{do}\ \mathit{Com} \\
 & & \mid l\colon \texttt{if}\ \mathit{Bexp}\ \texttt{then}\ \mathit{Com}\ \texttt{else}\ \mathit{Com}
\end{array}
$$

Fig. 1. Syntax of SPL (terms in square brackets are optional).

Now we fix a simple, sequential language (called SPL from now on) whose programs we analyze. The language allows local and global variables and recursion. For brevity, we assume that procedures do not take parameters or return values; these features are encoded using global variables.

The syntax of programs *Prog* and commands *Com* of SPL is as in Fig. 1. Here, `p` is a procedure name, `x` is a variable, l is a label, and *Aexp*, *Bexp* and *AConst* respectively stand for arithmetic and boolean expressions, and arithmetic constants. We restrict ourselves to *well-formed* programs where each label appears at most once. From now on, we assume an arbitrary but fixed program P.

The set of global variables in P is denoted by GV, and the set of local variables in a procedure $\mathtt{p}$ is denoted by $LV(\mathtt{p})$. The set of procedures is denoted by $Proc(P)$ or simply $Proc$. For each procedure $\mathtt{p}$, we denote by $Labels(\mathtt{p})$ the set of labels appearing in $\mathtt{p}$; this set contains a special label $\bot_{\mathtt{p}}$ that is reached when $\mathtt{p}$ terminates. The *first* label executed when $\mathtt{p}$ is run is denoted by $First(\mathtt{p})$.

We use a standard definition of the interprocedural control-flow graph (CFG) of P. Nodes here are labels of P. The edges are of three types: *call edges*, *local edges*, and *summary edges*. To define these, we construct a relation $Flow(\mathtt{p})$ between the labels of $\mathtt{p}$. If $(l, b, l') \in Flow(\mathtt{p})$ and l does not label a procedure call, then execution in $\mathtt{p}$ proceeds from l to l' if the guard b is true. If l is the "last" label in $\mathtt{p}$, then $(l, tt, \bot_{\mathtt{p}}) \in Flow(\mathtt{p})$. If l labels a call, then l' is the label to which the called procedure returns control on termination.

A *call edge* from procedure $\mathtt{p}$ to procedure $\mathtt{q}$ is now defined as a directed edge $e = (l, m)$, where $m = First(\mathtt{q})$ and l is the label of a command calling $\mathtt{q}$. A *local edge* $e = (l, b, m)$ in the procedure $\mathtt{p}$ goes from l to m (both l and m are labels in $\mathtt{p}$), and exists only if l does not label a procedure call and $(l, b, m) \in Flow(\mathtt{p})$. A *summary edge* $e = (l, \mathtt{q}, m)$ in $\mathtt{p}$ goes from l to m, and exists only if l labels a call to a procedure $\mathtt{q}$, and $(l, tt, m) \in Flow(\mathtt{p})$.

```
global flag, n

proc inc_n (): void = ...

proc bar() = local cond:=true
L1: while (cond) do
L2:  flag:=true;
L3:  if (*) then (L4: inc_n()) else
 (L5: flag:=false; L6: cond:=false)

proc main() =
L7: flag:=false; L8: n:=0;
L9: while (true) do
       (L10: bar(); L11: inc_n())
```

Fig. 2. Flagging and unflagging

The sets of call, local, and summary edges in the CFG of P are respectively denoted by E_{call}, E_{loc}, and E_{sum}. Finally, we define the *restriction* $P_{\mathtt{p}}$ of a program P with respect to a procedure $\mathtt{p}$ as the program obtained by removing from P all procedures unreachable from $\mathtt{p}$ in the CFG of P.

Figure 2 shows a program with procedures `main` and `bar`. The procedure `bar` need not terminate, but that if it does, it sets the flag to `false` before doing so.

Nested execution semantics. Now we give a semantics for SPL programs using nested words. Let us fix a set *Val* from which the values of our variables are drawn, and a special variable *pc* that captures the program counter and does not appear in the text of any of our programs. Now we define a *state* of a procedure $\mathtt{p}$ to be a map σ such that $\sigma(pc)$ is a label in $\mathtt{p}$, and for each $\mathtt{x} \in GV \cup LV(\mathtt{p})$, $\sigma(\mathtt{x}) \in Val$. An *entry state* of a procedure $\mathtt{p}$ is a state σ such that $\sigma(pc) = First(\mathtt{p})$, and for each local variable $\mathtt{u}$ of $\mathtt{p}$, we have $\sigma(\mathtt{u}) = n$ if $\mathtt{u}$ is initialized to n in $\mathtt{p}$. We denote the set of states of $\mathtt{p}$ by $States(\mathtt{p})$, and the set of states in P by $States$.

Note that a state as defined above does not contain a procedure stack. Let a *nested execution* now be a finite or infinite nested word over $States$. Our semantics assigns, to each procedure $\mathtt{p}$ in P, a set of nested executions.

Let a state σ of $\mathtt{p}$ be a *call state*, calling a procedure $\mathtt{q}$, if $\sigma(pc)$ is the label of a call to $\mathtt{q}$. For a call state σ of $\mathtt{p}$ calling $\mathtt{q}$, $Entry(\sigma, \mathtt{q})$ denotes the state

$\sigma_{en} \in \mathit{States}(\mathtt{q})$ such that: (1) $\sigma_{en}(pc) = \mathit{First}(\mathtt{q})$; (2) for each $\mathtt{g} \in \mathit{GVar}(P)$, we have $\sigma_{en}(\mathtt{g}) = \sigma(\mathtt{g})$; (3) for each local variable $\mathtt{u}$ of $\mathtt{q}$ initialized to n, we have $\sigma_{en}(\mathtt{u}) = n$. Intuitively, this is the entry state of $\mathtt{q}$ that is reached when $\mathtt{q}$ is called from the state σ. Likewise, for each call state σ_{call} of $\mathtt{p}$ that calls $\mathtt{q}$, and state $\sigma_{ex} \in \mathit{States}(\mathtt{q})$ such that $\sigma_{ex}(pc) = \bot_{\mathtt{q}}$, we define a "return state" $\mathit{Retn}(\sigma_{call}, \sigma_{ex})$ of $\mathtt{p}$ where control returns from the call.

The semantics of a procedure $\mathtt{p}$ is now defined using sets $[\![\mathtt{p}]\!]^*$ and $[\![\mathtt{p}]\!]^\omega$ respectively comprising its finite and infinite executions. The semantics of $\mathtt{p}$ is the union of these sets. We define these using sets $[\![c]\!]^*_{\mathtt{p}}$ and $[\![c]\!]^\omega_{\mathtt{p}}$, respectively comprising the finite and infinite executions of each command c in $\mathtt{p}$.

As $[\![\mathtt{p}]\!]^*$ and $[\![c]\!]^*_{\mathtt{p}}$ only contain terminating executions, they can be obtained by finite unrolling of loops and recursion. Accordingly, we define them as the *least fixpoint* of equations following the syntax of $\mathtt{p}$ and c. We only show a few cases:

1. $[\![l\colon \mathtt{x} := \mathit{exp}]\!]^*_{\mathtt{p}}$ comprises all $\sigma.\sigma'$ where $\sigma(pc) = l$, and σ' is obtained by taking σ and setting pc to $\bot_{\mathtt{p}}$ and $\mathtt{x}$ to the value of the expression exp in σ.
2. If c is a procedure call of the form $l\colon \mathtt{q}()$, then $[\![c]\!]^*_{\mathtt{p}} = L$, where L is the set of words $w' = \sigma.\langle.\sigma_{en}.w.\sigma_{ex}.\rangle.\sigma'$ such that: (1) $\sigma, \sigma' \in \mathit{States}(\mathtt{p})$ and $\sigma(pc) = l$; (2) $\sigma_{en} = \mathit{Entry}(\sigma, \mathtt{q})$; (3) $\sigma_{en}.w.\sigma_{ex} \in [\![\mathtt{q}]\!]^*$; and (4) $\sigma' = \mathit{Retn}(\sigma_{ex}, \sigma)$.
3. If the procedure $\mathtt{p}$ has the command c as its body, then $[\![\mathtt{p}]\!]^* = [\![c]\!]^*_{\mathtt{p}} \cap L_{En}(\mathtt{p})$ where $L_{En}(\mathtt{p})$ is the set of nested words over *States* starting with an entry state of $\mathtt{p}$.

Infinite nested executions of procedures and commands are defined similarly, except: (1) for commands that terminate—e.g., assignments—the set of infinite executions is empty; and (2) we have to take greatest fixpoints to define the semantics of loops and procedure calls. Finally, we define the notion of *local reachability* between states. For $\sigma, \sigma' \in \mathit{States}(\mathtt{p})$, σ' is *locally reachable* from σ (written as $\sigma \rightsquigarrow \sigma'$) if for some nested execution $w \in [\![\mathtt{p}]\!]$ and positions i and $j \geq i$, we have $w(i) = \sigma$, $w(j) = \sigma'$, and the word w_{ij} is matched.

4 Local invariants and summaries

Now we develop a class of invariants, called *local invariants*, that apply only to execution fragments within a single procedural context. To derive them, we use *procedure summaries* and reason with respect to environment assumptions.

We start by fixing an assertion language $\mathcal{A}$ and defining an *extended state* of a procedure $\mathtt{p}$ to be a pair (σ_{en}, σ) of states of $\mathtt{p}$. Intuitively, in an extended state (σ_{en}, σ), σ is the current state, and σ_{en} is the state at the beginning of the current procedural context. An *extended state formula* φ over $\mathtt{p}$ is an assertion in $\mathcal{A}$ such that φ may use two free variables x_{en} and x for each variable (including the control variable pc) $\mathtt{x}$ in scope in $\mathtt{p}$.[1]

[1] As a convention, we use typewriter font to refer to program variables, and italics to refer to logical variables.

Such a formula is interpreted over extended states (σ_{en}, σ), with x_{en} and x capturing the values of $\mathtt{x}$ at σ_{en} and σ; every formula thus encodes a set of extended states. We write $(\sigma_{en}, \sigma) \models \varphi$ if (σ_{en}, σ) satisfies φ. If all extended states satisfy φ, then we write $\models \varphi$. Also, we denote the set of extended state formulas over $\mathtt{p}$ by $Assn(\mathtt{p})$.

A *local invariant* of $\mathtt{p} \in Proc$ is a formula $\pi \in Assn(\mathtt{p})$ such that for any nested execution $w \in [\![\mathtt{p}]\!]$, the local path w_l of w satisfies the following property: for all positions i in w_l, $(w_l(0), w_l(i)) \models \pi$. A *summary* of a procedure $\mathtt{p}$ is a formula $\psi \in Assn(\mathtt{p})$ such that for each finite nested execution $w \in [\![\mathtt{p}]\!]$ ending at a position n, $(w(0), w(n)) \models \psi$. Intuitively, local invariants assert conditions that hold on the path through the "top-level" context of a nested execution. Note that the formula $(\pi \wedge (pc = \bot_{\mathtt{p}}))$ is a summary of the procedure $\mathtt{p}$ if π is a local invariant of $\mathtt{p}$.

Inductive local invariants and summaries. Our goal here is to obtain, for each procedure $\mathtt{p}$, an *inductive local invariant*. This is done with respect to a summary of each procedure called from $\mathtt{p}$. Due to recursion, these invariants and summaries may be interdependent, and need to be defined via mutual induction.

These notions are developed via a simple generalization of the non-procedural case. First we define a *predicate transformer* for each edge e in the CFG of P. For a local edge $e = (l, b, m)$ in the procedure $\mathtt{p}$, such a transformer takes a formula $\varphi \in Assn(\mathtt{p})$, and returns a formula $\varphi' = Post_e(\varphi) \in Assn(\mathtt{p})$ that encodes the least set S of extended states such that for each (σ_{en}, σ) that satisfies φ and is such that $\sigma(pc) = l$, if σ' can be reached by executing the statement (l, b, m) from σ, then $(\sigma_{en}, \sigma') \in S$. We write $\{\varphi\}\ e\ \{\varphi'\}$ if $Post_e(\varphi) \Rightarrow \varphi'$.

Predicate transformers for call edges e are similar, except for $\varphi \in Formulas(\mathtt{p})$, $Post_e(\varphi) \in Formulas(\mathtt{q})$, where $\mathtt{q}$ is a procedure called from $\mathtt{p}$. If e is a summary edge capturing execution within a called procedure $\mathtt{q}$, then its predicate transformer takes in a *summary* ψ of $\mathtt{q}$ as an extra parameter, and is of the form $Post_e(\varphi, \psi)$. Here, for given φ and ψ, $\varphi' = Post_e(\varphi, \psi)$ represents the least set of extended states S such that if (σ_{en}, σ) satisfies φ and σ is a call to procedure $\mathtt{q}$, then assuming the summary ψ for $\mathtt{q}$ and the return state σ_{ret}, we have $(\sigma_{en}, \sigma_{ret}) \in S$. Again, we write $\{\varphi\}\ (e, \psi)\ \{\varphi'\}$ if $Post_e(\varphi, \psi) \Rightarrow \varphi'$. We omit the detailed encodings of these formulas.

Finally, let us define a formula $\mathcal{I}_{\mathtt{p}}$ capturing the *initial condition* of a procedure $\mathtt{p}$ (details omitted). Inductive local invariants and summaries are now defined as follows:

Definition 1. *Let P have procedures $\mathtt{p_1}, \ldots, \mathtt{p_k}$ and initial procedure $\mathtt{p_{in}}$. The* inductive local invariant *and* summary *for each procedure $\mathtt{p_i}$ are respectively given by $I(\mathtt{p_i})$ and $\Psi(\mathtt{p_i})$, where I and Ψ are maps that assign an extended state formula to each procedure in P, and satisfy the following:*

1. $\models \mathcal{I}_{\mathtt{p_{in}}} \Rightarrow I(\mathtt{p_{in}}) \wedge (pc = pc_{en} = First(\mathtt{p_{in}}))$
2. *for each local edge $e = (l, b, m)$ in $\mathtt{p}$,* $\models \{I(\mathtt{p}) \wedge (pc = l)\}\ e\ \{I(\mathtt{p}) \wedge (pc = m)\}$
3. *for each summary edge $e = (l, \mathtt{q}, m)$ in $\mathtt{p}$,*
   $\models \{I(\mathtt{p}) \wedge (pc = l)\}\ (e, \Psi(\mathtt{q}))\ \{I(\mathtt{p}) \wedge (pc = m)\}$

4. *for each call edge* $e = (l, m)$ *from* $\mathtt{p}$ *to* $\mathtt{q}$,
   $\models \{I(\mathtt{p}) \wedge (pc = l) \wedge \mathcal{I}_{\mathtt{q}}\}\ e\ \{I(\mathtt{q}) \wedge (pc = \mathit{First}(\mathtt{q}))\}$
5. *for all* $\mathtt{p}$, *we have* $\models I(\mathtt{p}) \wedge (pc = \bot_{\mathtt{p}}) \Rightarrow \Psi(\mathtt{p})$.

A pair (I, Ψ) *of maps as above is called an* inductive pair.

Intuitively, condition (1) requires that the inductive local invariant, when asserted at the label where the program starts execution, satisfies the initial conditions of $\mathtt{p_{in}}$. Conditions (2) and (3) require that invariants are preserved under transitions along local and summary edges. Condition (4) asserts the initial conditions of a procedure at its entry states reached via calls. Condition (5) relates summaries given by Ψ to invariants given by I.

It is not hard to show that Definition 1 is sound:

Lemma 1. *If* (I, Ψ) *is an inductive pair, then for each* $\mathtt{p} \in \mathit{Proc}$, $I(\mathtt{p})$ *is a local invariant and* $\Psi(\mathtt{p})$ *a summary of* $\mathtt{p}$.

For example, consider the program in Figure 2. Suppose, assuming `inc_n` only increments `n`, we want to derive the local invariant (*flag* = *ff*) for `main`. The required reasoning is performed in a procedure-modular way. First we just consider the body of `main`, while making the necessary *assumptions* about the procedures it calls (in this case, `bar`). We note that the invariant holds if (*flag* = *ff*) is a summary for `bar`. Now we must validate this summary by reasoning about `bar`. Here we assume the invariant (*cond* $\vee$ (*flag* = *ff*)) for the label *L2* and show that this is a loop invariant. Verifying the summary is now easy.

5 Temporal verification

Local invariants may be directly applied in proving temporal safety and liveness properties interpreted on nested program executions. We explore three classes of temporal properties—*safety*, *response*, and *reactivity*—each of which has three subclasses corresponding to interpretations on local, global, and staircase paths in nested executions. Of these, staircase reactivity properties can capture all properties expressible in temporal logic over nested words [13, 6].

In the following, we write $P, \mathtt{p} \models f$ if the procedure $\mathtt{p}$ in the program P satisfies a temporal property f (we will define what this means for each property we consider). We write $P, \mathtt{p} \vdash_{\mathsf{R}} f$, often omitting P and/or R, if we can prove using a rule R that $\mathtt{p}$ satisfies f. Finally, we write $\vdash \varphi$ if we can prove the extended state formula φ.

A rule R proving a property f of a procedure $\mathtt{p}$ in a program P is called *sound* we have $P, \mathtt{p} \vdash_{\mathsf{R}} f$ only if $P, \mathtt{p} \models f$. As for completeness, consider sets $S_1, \ldots, S_k$ of extended states. We call R *complete relative to these sets* if, assuming that each S_i can be encoded by an extended state formula and that all assertions in $\mathcal{A}$ can be proved or disproved, we have $P, \mathtt{p} \models f$ only if $P, \mathtt{p} \vdash_{\mathsf{R}} f$. We call R *relatively complete* if it is complete relative to a collection of sets of extended states, each of which can be captured using $\mathcal{A}$.

Local safety. A local safety property says: "In any nested execution of a procedure $\mathtt{p}$, a certain assertion is never violated in the top-level procedural context." We define:

Definition 2. *Let $\varphi \in Assn(\mathtt{p})$ for a procedure $\mathtt{p}$. The procedure $\mathtt{p}$ satisfies the* local safety property $\Box^l\varphi$ *(read as "Always locally φ") if for each $w \in [\![\mathtt{p}]\!]$ and for each position i in the local path $\sigma_0\sigma_1\ldots$ in w, (σ_0, σ_i) satisfies φ. This fact is written as $P, \mathtt{p} \models \Box^l\varphi$)*

Input: (1) Procedure $\mathtt{p}$ in program P; (2) $\varphi \in Assn(\mathtt{p})$

Rule: Find an inductive pair (I, Ψ) for the program $P_\mathtt{p}$ such that $\vdash I(\mathtt{p}) \Rightarrow \varphi$

$$P, \mathtt{p} \vdash \Box^l\varphi$$

Fig. 3. Rule L-SAFE for local safety

Fig. 3 shows our rule L-SAFE for local safety. The rule is a generalization of the classic proof rule for temporal safety [15]. Unlike in the classical case, the inductive invariant we need here is a *local* invariant. To prove local safety for $\mathtt{p}$, we only need to consider the program $P_\mathtt{p}$.

Example 1. Recall the program in Fig. 2, and consider the safety property: "`flag` is always false." While this property is violated by global program executions, it holds locally in `main`. A proof follows from the inductive pair for this program derived earlier. In fact, this example represents a class of applications of local safety properties: those where an invariant may be legitimately broken by a called procedure, so long as it is restored before control returns.

Soundness of L-SAFE follows from Lemma 1:

Theorem 1. *The rule* L-SAFE *is sound.*

As for completeness, let $Proc(P_\mathtt{p})$ be the set of procedures in $P_\mathtt{p}$, and let $S^R_\mathtt{q}$ be, for each $\mathtt{q} \in Proc(P_\mathtt{p})$), the set of extended states (σ_{en}, σ) such that σ_{en} is an entry state of $\mathtt{q}$ and $\sigma_{en} \rightsquigarrow \sigma$. Thus, the set $S^R_{\mathtt{q}_\mathtt{i}}$ captures local reachability from an entry state of $\mathtt{q}$. We have:

Theorem 2. L-SAFE *is complete relative to the sets $S^R_\mathtt{q}$, where $\mathtt{q} \in Proc(P_\mathtt{p})$.*

Proof: Let us assume that $P, \mathtt{p} \models \Box^l\varphi$. For each $\mathtt{q} \in Proc(P_\mathtt{p})$, let $\chi_\mathtt{q}$ be an extended state formula capturing the set $S^R_\mathtt{q}$ (i.e., for each extended state (σ_{en}, σ) of $\mathtt{q}$, we have $(\sigma_{en}, \sigma) \models \chi_\mathtt{q}$ iff $(\sigma_{en}, \sigma) \in S^R_\mathtt{q}$). By our assumption, these formulas exist. Now consider the pair of maps (I, Ψ), each assigning a formula to each $\mathtt{q}$ as above, such that for all such $\mathtt{q}$, we have $I(\mathtt{q}) = \chi_\mathtt{q}$ and $\Psi(\mathtt{q}) = I(\mathtt{q}) \wedge (pc = \bot_\mathtt{q})$.

We claim that (I, Ψ) is an inductive pair for $P_\mathtt{p}$. To see why this is so, consider the conditions in Definition 1. Condition (1) holds because $(\sigma_{in}, \sigma_{in})$, where σ_{in}

is an entry state of $\mathtt{p}$ belongs to $S^R_{\mathtt{p}}$. Condition (5) is similarly verified, and condition (6) holds trivially from our choice of Ψ. Conditions (2), (3), and (4) follow from the definition of local reachability and predicate transformers, and the hypothesis that Ψ captures summaries.

Now note that $I(\mathtt{p}) \Rightarrow \varphi$. Recall that $(\sigma_{en}, \sigma) \models \varphi$ for all entry states σ_{en} of $\mathtt{p}$ and all σ such that $\sigma_{en} \rightsquigarrow \sigma$. As $I(\mathtt{p})$ (i.e., $\chi_{\mathtt{p}}$) precisely characterizes those pairs, (I, Ψ) satisfies the premises of L-SAFE. Thus, $P, \mathtt{p} \vdash \Box^l \varphi$. □

Now we show a way to encode the sets $S^R_{\mathtt{q}}$ using assertions, generalizing a technique in Manna and Pnueli's completeness proof [14] and proving that:

Theorem 3. L-SAFE *is relatively complete.*

Proof: We assume that our data domain can express records and *binary trees* of records; our assertions use auxiliary variables of these types. For a node u in a tree τ of records, let $lc(u)$ and $rc(u)$ respectively denote the left and right children of u (the right child may not exist, in which case we write $rc(u) = \perp$). The root of τ is denoted by $root(\tau)$; u satisfies the predicate $leaf(u)$ iff it is a leaf.

The records u forming the tree nodes have fields indexed by the logical variables x_{en} and x of our state formulas. For an extended state formula ψ, the *application* $\psi(u)$ is obtained by substituting the free variables of ψ with the corresponding fields of u. The formula $\widetilde{V} = u$ has free variables x and x_{en} for every variable $\mathtt{x}$ of $\mathtt{q}$, and states that each free variable has the value of the corresponding field in u. For each local or call edge e, $Post_e(u)$ refers to $Post_e(\psi_u)$, where ψ_u states that each variable has the value of the corresponding field in u. The application of $Post_e(u)$ to a node u' is denoted by $(u = Post_e(u'))$. If e is a summary edge, the formula $(u = Post_e(u', u''))$ (where u', u'' are records) is likewise defined.

The formula $\chi_{\mathtt{q}}$ is:

$$\chi_{\mathtt{q}} : \exists \tau.((|\tau| > 0) \wedge \lambda_{leaf} \wedge \lambda_{root} \wedge \forall u.(\neg leaf(u) \Rightarrow \delta_{loc} \vee \delta_{sum}))$$

where

$$\begin{array}{ll}
\lambda_{leaf} : & \forall u.\ leaf(u) \Rightarrow \bigvee_{\mathtt{r} \in Proc} (\mathcal{I}_{\mathtt{q}} \wedge (pc = pc_{en} = First(\mathtt{q}))(u) \\
\lambda_{root} : & \widetilde{V} = root(\tau) \\
\delta_{loc} : & (rc(u) = \perp) \wedge \bigvee_{e \in E_{loc}} (u = Post_e(lc(u))) \\
\delta_{sum} : & (rc(u) \neq \perp) \wedge \bigvee_{e \in E_{sum}} (u = Post_e(lc(u), rc(u)))
\end{array}$$

The assertion $\chi_{\mathtt{p}}$ encodes a proof tree establishing local reachability between states σ_{en} and σ in $\mathtt{p}$ (also, σ_{en} is an entry state of $\mathtt{p}$). The root of τ encodes variable values at these states. The leaves encode the fact that for each state σ, we have $\sigma \rightsquigarrow \sigma$. The children of a node $u = (\sigma'_{en}, \sigma')$ capture reachability facts that, together, imply that σ' is locally reachable from σ'_{en} (note that these states are not necessarily in $\mathtt{p}$; also, if u has no right child, then only one premise is needed to derive it). For example, u may have a single child (σ'_{en}, σ''), where σ'' has a transition along a local edge to σ'. Thus, $\chi_{\mathtt{p}}$ captures $S^R_{\mathtt{p}}$. □

Input: (1) Procedure $\mathtt{p}$ in program P; (2) Formulas $\varphi_1, \varphi_2 \in \mathit{Formulas}(\mathtt{p})$

Rule: Find an inductive pair (I, Ψ) for the program $P_{\mathtt{p}}$, a ranking function from extended states of P to D, a formula $\kappa \in \mathit{Formulas}(\mathtt{p})$ and, for each procedure $\mathtt{q} \in \mathit{Proc}(P_{\mathtt{p}})$, a formula $\beta_{\mathtt{q}} \in \mathit{Assn}(\mathtt{q})$, such that:

1. $\vdash \varphi_1 \Rightarrow \varphi_2 \vee \kappa$;
2. For each local edge e in $\mathtt{p}$,
   $\vdash \{\kappa \wedge (\delta = d)\}\ e\ \{\varphi_2 \vee (\kappa \wedge (\delta \prec d)\}$;
   for each local edge in a procedure $\mathtt{q}$,
   $\vdash \{\beta_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{\{(pc = \bot_{\mathtt{q}}) \vee (\beta_{\mathtt{q}} \wedge (\delta \prec d)\}$
3. For each call edge e from $\mathtt{p}$ to a procedure $\mathtt{q}$,
   $\vdash \{\kappa \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{q}} \wedge (\delta \prec d)\}$;
   for each call edge from a procedure $\mathtt{q}$ to a procedure $\mathtt{r}$,
   $\vdash \{\beta_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{r}} \wedge (\delta \prec d)\}$
4. For each summary edge $e = (l, \mathtt{r}, m)$ in $\mathtt{p}$,
   $\vdash \{\kappa \wedge (\delta = d)\}\ (e, \Psi(\mathtt{r}))\ \{\varphi_2 \vee (\kappa \wedge (\delta \prec d))\}$;
   for each such edge in a procedure $\mathtt{q}$,
   $\vdash \{\beta_{\mathtt{q}} \wedge (\delta = d)\}\ (e, \Psi(\mathtt{r}))\ \{(pc = \bot_{\mathtt{q}}) \vee (\beta_{\mathtt{q}} \wedge (\delta \prec d))\}$

$$P, \mathtt{p} \vdash \Box^l(\varphi_1 \Rightarrow \Diamond^l \varphi_2)$$

Fig. 4. Rule L-RESP for local response

Local response. Now we extend our approach to liveness. We define *local response* as:

Definition 3. *Let $\varphi_1, \varphi_2 \in \mathit{Assn}(\mathtt{p})$ for a procedure $\mathtt{p}$. The procedure $\mathtt{p}$ satisfies the local response property $f = \Box^l(\varphi_1 \Rightarrow \Diamond^l \varphi_2)$ if for each $w \in [\![\mathtt{p}]\!]^\omega$ and for each position i in the local path $\sigma_0\sigma_1 \ldots$ such that $(\sigma_0, \sigma_i) \models \varphi_1$, there exists $j \geq i$ such that $(\sigma_0, \sigma_j) \models \varphi_2$. This fact is written as $P, \mathtt{p} \models f$.*

Note that the definition only considers the *infinite* executions of $\mathtt{p}$.

Liveness properties as above are proved by generalizing techniques from classical verification using ranking functions. Let $(D, \preceq)$ be a well-founded preorder; for $a, b \in D$, we write $a = b$ if $a \preceq b$ and $b \preceq a$, and $a \prec b$ if $a \preceq b$ and $a \neq b$. Let a *ranking function* for the above preorder and the program P be a map $\delta : (\sigma_{en}, \sigma) \mapsto d$, where (σ_{en}, σ) is an extended state and $d \in D$. We use extended state formulas such as $(\delta \preceq d)$ and $(\delta = d)$ that are satisfied by an extended state (σ_{en}, σ) respectively when $\delta(\sigma_{en}, \sigma) \preceq d$ and $\delta(\sigma_{en}, \sigma) = d$. Ways to encode such assertions in a language like $\mathcal{A}$ may be found in [14].

Our rule L-RESP for local response is in Fig. 4. Intuitively, the obligation κ is asserted whenever φ_1 holds along a local path, and is "released" only when φ_2 holds on this path as well. In path fragments where κ is asserted, the ranking function decreases in value; as D has no infinite descending chain, this means that φ_2 will hold eventually.

Now, when the execution enters a new context via a call, the execution fragment from then on till the matching return is not part of the local path. Suppose κ was not released by the time the call happened. If the call never terminates, the local path will have ended at the call, and the response property will be violated.

Input: (1) Procedure $\mathtt{p}$ in program P; (2) Formulas $\varphi_1, \varphi_2 \in \mathit{Formulas}(\mathtt{p})$

Rule: Find an inductive pair (I, Ψ) for the program $P_{\mathtt{p}}^{\varphi_2}$, a ranking function from extended states of P to D, and, for each procedure $\mathtt{q}$ in $P_{\mathtt{p}}^{\varphi_2}$, a formula $\kappa_{\mathtt{q}} \in \mathit{Assn}(P)$, such that:

1. $\vdash (pc = l) \wedge \varphi_1 \Rightarrow (\varphi_2 \vee \kappa_{\mathtt{q}})$, if the label l is in $\mathtt{q}$;
2. For each local edge e in a procedure $\mathtt{q}$,
   $\vdash \{\kappa_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{\varphi_2 \vee (\kappa_{\mathtt{q}} \wedge (\delta \prec d)\}$;
3. For each call edge from procedure $\mathtt{q}$ to procedure $\mathtt{r}$,
   $\vdash \{\kappa_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{\varphi_2 \vee (\kappa_{\mathtt{r}} \wedge (\delta \prec d))\}$
4. For each summary edge $e = (l, \mathtt{r}, m)$ in procedure $\mathtt{q}$,
   $\vdash \{\kappa_{\mathtt{q}} \wedge (\delta = d)\}\ (e, \Psi(\mathtt{r}))\ \{\neg\#_{\varphi_2} \Rightarrow (\varphi_2 \vee (\kappa_{\mathtt{q}} \wedge (\delta \prec d)))\}$

$$P, \mathtt{p} \vdash \Box^g(\varphi_1 \Rightarrow \Diamond^g \varphi_2)$$

Fig. 5. Rule G-RESP for global response

Consequently, we must ensure that all such calls eventually return. This is done using the properties $\beta_{\mathtt{q}}$ (split among procedures), which are just like κ, except they are released when the "terminal" label $\bot_{\mathtt{q}}$ is reached. Note that because of recursive calls, a procedure may be re-entered—e.g., we may have $\mathtt{q} = \mathtt{p}$.

Example 2. In the program in Fig. 2, suppose we want to show that `bar` satisfies the property $\Box^l(\mathit{cond} \Rightarrow \Diamond^l(\neg \mathit{flag} \vee (n \geq n_{en} + 100)))$. This is done using a ranking function that maps each extended state (σ_{en}, σ) of `bar` to a pair (l, v), where l is the label of σ, and v is the value of $\max\{0, (n_{en} + 100 - n)\}$ in this extended state. The labels are partially ordered as $(L1 < L2 < L3)$, $(L4 < L3)$, and $(L5 < L3)$. We have $(l', v') \prec (l, v)$ iff either $(v' < v)$, or $(v' = v)$ and $(l' < l)$.

Now κ says: "*pc* is one of *L1*, *L2*, *L3*, *L4*, or *L5*, and $(n < n_{en} + 100)$." Clearly, this satisfies the rule's premises.

We can show that:

Theorem 4. *The rule* L-RESP *is sound and relatively complete.*

Global response. Local invariants may also be used to modularly prove properties of executions spanning multiple contexts. The simplest of these is *global safety*. Here we consider the *global response property* $\Box^g(\varphi_1 \Rightarrow \Diamond^g \varphi_2)$, which is defined in exactly the same way as local response, except that it is interpreted on the global rather than the local path.

Our rule G-RESP for global response is in Fig. 5) To understand it, first consider the rule for local response and a state of procedure $\mathtt{p}$ that calls the procedure $\mathtt{q}$ and satisfies κ, but not φ_2. Clearly, this state was reached along a local path where φ_1 held at one point, but φ_2 has not held since. In local response, we had to ensure that this call terminates, and that φ_2 holds along the local path in the continuation. In global response, we do not need termination: a non-returning path is legitimate if φ_2 eventually holds in it. However, we must

Input: (1) Procedure $\mathtt{p}$ in program P; (2) Formulas $\varphi_1, \varphi_2, \theta \in \mathit{Formulas}(P)$

Rule: Find an inductive pair (I, Ψ) for the program $P_{\mathtt{p}}$, a ranking function from extended states of P to D, a formula $\kappa \in \mathit{Formulas}(\mathtt{p})$ and, for each procedure $\mathtt{q} \in \mathit{Proc}(P_{\mathtt{p}})$, a formula $\beta_{\mathtt{q}} \in \mathit{Assn}(\mathtt{q})$, such that:

1. $\vdash \varphi_1 \Rightarrow \varphi_2 \vee \kappa$;
2. For each local edge e in $\mathtt{q}$,
   $\vdash \{\kappa \wedge \theta \wedge (\delta = d)\}\ e\ \{\varphi_2 \vee (\kappa \wedge (\delta \prec d)\}$ $\quad \vdash \{\kappa \wedge (\delta = d)\}\ e\ \{\varphi_2 \vee (\kappa \wedge (\delta \preceq d)\}$
   $\vdash \{\beta_{\mathtt{q}} \wedge \theta \wedge (\delta = d)\}\ e\ \{(pc = \bot_{\mathtt{q}}) \vee \varphi_2 \vee (\beta_{\mathtt{q}} \wedge (\delta \prec d)\}$
   $\vdash \{\beta_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{(pc = \bot_{\mathtt{q}}) \vee \varphi_2 \vee (\beta_{\mathtt{q}} \wedge (\delta \preceq d)\}$
3. For each call edge e from a procedure $\mathtt{q}$ to a procedure $\mathtt{r}$,
   $\vdash \{\kappa \wedge \theta \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{r}} \wedge (\delta \prec d)\}$ $\quad \vdash \{\kappa \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{r}} \wedge (\delta \preceq d)\}$
   $\vdash \{\beta_{\mathtt{q}} \wedge \theta \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{r}} \wedge (\delta \prec d)\}$ $\quad \vdash \{\beta_{\mathtt{q}} \wedge (\delta = d)\}\ e\ \{\beta_{\mathtt{r}} \wedge (\delta \preceq d)\}$
4. For each summary edge $e = (l, \mathtt{q}, m)$ in a procedure $\mathtt{r}$,
   $\vdash \{\kappa \wedge \theta \wedge (\delta = d)\}\ (e, \Psi(\mathtt{q}))\ \{\varphi_2 \vee (\kappa \wedge (\delta \prec d))\}$
   $\vdash \{\kappa \wedge (\delta = d)\}\ (e, \Psi(\mathtt{q}))\ \{\varphi_2 \vee (\kappa \wedge (\delta \preceq d))\}$
   $\vdash \{\beta_{\mathtt{r}} \wedge \theta \wedge (\delta = d)\}\ (e, \Psi(\mathtt{q}))\ \{(pc = \bot_{\mathtt{r}}) \vee \varphi_2 \vee (\beta_{\mathtt{r}} \wedge (\delta \prec d))\}$
   $\vdash \{\beta_{\mathtt{r}} \wedge (\delta = d)\}\ (e, \Psi(\mathtt{q}))\ \{(pc = \bot_{\mathtt{r}}) \vee \varphi_2 \vee (\beta_{\mathtt{r}} \wedge (\delta \preceq d))\}$

$$P, \mathtt{p} \vdash \Box^s((\varphi_1 \wedge \Box^s \Diamond^s \theta) \Rightarrow \Diamond^s \varphi_2)$$

Fig. 6. Rule S-REACT for staircase reactivity

assert that in all executions that do reach the matching return without having satisfied φ_2 in the interim, an invariant like κ must be asserted at the matching return. This requires us to relate the fragment of the execution within $\mathtt{q}$ with the conditions that hold afterwards. It is possible to do this using an auxiliary program variable.

For an assertion φ and a program P, let us define the program P^{φ} obtained by modifying P as follows. To each procedure $\mathtt{p}$ of P, we add a local boolean variable $\#_{\mathtt{p},\varphi}$. Between every two commands in $\mathtt{p}$, we add the command `if(`φ`) then (`$\#_{\mathtt{p},\varphi}$`:=true) else skip.` We also make $\mathtt{p}$ return the value of this variable. This is encoded using a global variable γ—the last command in $\mathtt{p}$ stores the value of $\#_{\mathtt{p},\varphi}$ in γ. Finally, after each procedure call from $\mathtt{p}$ to $\mathtt{q}$, we add a statement $\#_{\mathtt{p},\varphi} = \gamma$.

This augmented program tracks if φ is satisfied within a procedure $\mathtt{q}$ called from $\mathtt{p}$. As $\mathtt{q}$ returns the value of $\#_{\mathtt{q},\varphi}$ on termination, we can refer to this value to see if φ was satisfied within the called context.

The rule G-RESP uses such an augmentation of the input program P. The interesting premise concerns summary edges: we assert liveness at the target of such an edge only if the procedure's auxiliary variable is false at that point (i.e., if the property is not satisfied within the context summarized by the edge).

Example 3. Consider the program in Fig. 2 once again, and the global response property $\Box^g((n = 0) \Rightarrow \Diamond^g(n \geq 1))$. While the local version of this property is not satisfied by the procedure `main`, the global version is easily verified using G-RESP. As `bar` may or may not terminate or not increment n, the auxiliary variables are critical to the proof.

Soundness and completeness are obtained by slightly modifying the corresponding proofs for local response:

Theorem 5. G-RESP *is sound and relatively complete.*

Staircase reactivity. Now we prove the most general of our properties: *staircase reactivity*. A staircase reactivity property asserts: "Along the staircase path in any nested execution, if φ_1 holds infinitely often, then φ_2 also holds infinitely often." These properties can capture the parity acceptance condition of ω-automata. As automata operating on the staircase path can capture all ω-regular properties of nested words [6], a complete rule for staircase reactivity can prove all temporal properties of nested executions.

Following [14], we use a syntactic formulation of reactivity that involves an extra assertion θ. We define:

Definition 4. *Let $\varphi_1, \varphi_2, \theta \in Assn(\mathtt{p})$ for a procedure* p. *The procedure* p *satisfies the* staircase reactivity property $f = \Box^s(\varphi_1 \wedge \Box^s\Diamond^s\theta \Rightarrow \Diamond^s\varphi_2)$ *if for each $w \in [\![\mathtt{p}]\!]$ and for each position i in the staircase path $\sigma_0\sigma_1\ldots$ such that: (1) $(\sigma_0, \sigma_i) \models \varphi_1$, and (2) there exist infinitely many $j \geq i$ such that $(\sigma_0, \sigma_j) \models \theta$, there is some $k \geq i$ such that $(\sigma_0, \sigma_k) \models \varphi_2$.*

Our rule S-REACT for staircase reactivity is shown in Fig. 6. The rule combines features of proofs for local and global properties, and generalizes the rule for response.

Consider, first, the case where there are no procedure calls. As in local response, κ is asserted whenever an extended state satisfying φ_1 is reached along a local path, and continues to hold till the "goal" of reaching φ_2 is met. However, this time the rank decreases along a path fragment with invariant κ only when θ is satisfied (and it never increases along a path). If θ holds infinitely often, then either φ_2 holds eventually, or the rank must decrease unboundedly. The latter is impossible as D is well-founded.

If the program has procedure calls, we propagate two liveness conditions at each call. Along the call edge, we assert the property that along each path within the new context, either the reactivity condition is met, or the matching return of the present call is reached. Along the summary edge, we assert: "the reactivity condition is met eventually."

To see why, suppose a call terminates *after having satisfied the liveness obligation*. The part of this execution within the called context is not in the staircase path, but this is not an issue as liveness is asserted along the summary edge *regardless of* what happens within the called context. Now suppose this call never returns. In this case, using a strong summary, we rule out a continuation of the current execution along the summary edge in question. However, the condition for the call edge ensures that the context reached via the call satisfies the liveness obligation. In general, we can show that:

Theorem 6 (Soundness, completeness). *The rule* S-REACT *is sound and relatively complete.*

References

1. R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Proceedings of LICS*, pages 151–160, 2007.
2. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proceedings of TACAS*, pages 467–481, 2004.
3. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of STOC*, pages 202–211, 2004.
4. R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, pages 1–13, 2006.
5. K. R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
6. M. Arenas, P. Barceló, and L. Libkin. Regular languages of nested words: Fixed points, automata, and synchronization. In *ICALP*, pages 888–900, 2007.
7. T. Ball and S. Rajamani. The SLAM toolkit. In *13th International Conference on Computer Aided Verification*, pages 260–264, 2001.
8. A. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
9. B. Cook, A. Podelski, and A. Rybalchenko. CFL-termination. Technical Report MSR-TR-2008-160, 2008.
10. I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, pages 270–280, 2008.
11. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of CAV*, pages 526–538, 2002.
12. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
13. C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *Proceedings of FSTTCS*, pages 408–420, 2004.
14. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):91–130, 1991.
15. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, New York, 1995.
16. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Progress*. 1996.
17. A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS*, pages 46–77, 1977.
18. A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In *ESOP*, pages 94–107, 2005.
19. T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of POPL*, pages 49–61, 1995.
20. T. W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of SAS*, pages 189–213, 2003.
21. M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.