Acta Informatica (2008) 45:403–439
DOI 10.1007/s00236-008-0075-2

ORIGINAL ARTICLE

# Inference rules for proving the equivalence of recursive procedures

**Benny Godlin · Ofer Strichman**

Received: 12 May 2006 / Revised: 21 September 2006 / Published online: 2 July 2008

**Abstract** Inspired by Hoare's rule for recursive procedures, we present three proof rules for the equivalence between recursive programs. The first rule can be used for proving partial equivalence of programs; the second can be used for proving their mutual termination; the third rule can be used for proving the equivalence of reactive programs. There are various applications to such rules, such as proving equivalence of programs after refactoring and proving backward compatibility.

## Contents

B. Godlin
Department of Computer Science, Technion, Haifa, Israel
e-mail: bgodlin@cs.technion.ac.il

O. Strichman (✉)
Information Systems, IE, Technion, Haifa, Israel
e-mail: ofers@ie.technion.ac.il

## 1 Introduction

We propose three proof rules for proving equivalence between possibly recursive programs, which are inspired by Hoare's rule for recursive procedures [7]. The first rule can be used for proving partial equivalence (i.e., equivalence if both programs terminate); the second rule can be used for proving mutual termination (i.e., one program terminates if and only if the other terminates); the third rule proves equivalence of reactive programs. Reactive programs maintain an ongoing interaction with their environment by receiving inputs and emitting outputs, and possibly run indefinitely (for example, an operating system is a reactive program). With the third rule we can possibly prove that two such programs generate equivalent sequences of outputs, provided that they receive equal sequences of inputs. The premise of the third rule implies the premises of the first two rules, and hence it can be viewed as their generalization. We describe these and other notions of equivalence more formally later in this section.

The ability to prove equivalence of programs can be useful in various scenarios, such as comparing the code before and after manual *refactoring*, to prove *backward compatibility*, as done by Intel for the case of microcode [1] (yet under the restriction of not supporting loops and recursions) and for performing what we call *regression verification*, which is a process of proving the equivalence of two closely-related programs, where the equivalence criterion is user-defined.

First, consider refactoring. To quote Martin Fowler [4,5], the founder of this field, "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring". The following example demonstrates the need for proving equivalence of recursive functions after an application of a single refactoring rule. A list of some of the refactoring rules that can be handled by the proposed rules is given in Appendix B.

*Example 1* The two equivalent programs in Fig. 1 demonstrate the *Consolidate Duplicate Conditional Fragments* refactoring rule. These recursive functions calculate the value of a given number `sum` after `y` years, given that there is some annual interest, which depends on whether there is a "special deal". The fact that the two functions return the same values given the same inputs, and that they mutually terminate, can be proved with the rules introduced in this article. □

Next, consider regression verification. Regression verification is a natural extension of the better-known term *regression testing*. Reasoning about the correctness of a program while using a previous version as a reference, has several distinct advantages over functional

```
int calc1(sum,y) {

  if (y <= 0) return sum;

  if (isSpecialDeal()) {
    sum = sum * 1.02;
    return calc1(sum, y-1);
  }
  else {
    sum = sum * 1.04;
    return calc1(sum, y-1);
  }
}
```

```
int calc2(sum,y) {

  if (y <= 0) return sum;

  if (isSpecialDeal())
    sum = sum * 1.02;
  else
    sum = sum * 1.04;
  return calc2(sum, y-1);
}
```

**Fig. 1** A refactoring example

verification of the new code (although both are undecidable in general). First, code that cannot be easily specified and verified can still be checked throughout the development process by examining its evolving effect on user-specified variables or expressions. Second, comparing two similar systems should be computationally easier than property-based verification, relying on the fact that large portions of the code has not changed between the two versions[1].

Regression verification is relevant, for example, for checking equivalence after implementing optimizations geared for performance, or checking side-effects of new code. For example, when a new flag is added, which changes the result of the computation, it is desirable to prove that as long as this flag is turned off, the previous functionality is maintained.

Formally verifying the equivalence of programs is an old challenge in the theorem-proving community (see some recent examples in [10–12]). The current work can assist such proofs since it offers rules that handle recursive procedures while decomposing the verification task: specifically, the size of each verification condition is proportional to the size of two individual procedures. Further, using the rules requires a decision procedure for a restricted version of the underlying programming language, in which procedures contain no loops or procedure calls. Under these modest requirements several existing software verification tools for popular programming languages such as C are complete. A good example of such a tool for ANSI-C is CBMC [8], which translates code with a bounded number of loops and recursive calls (in our case, none) to a propositional formula.[2]

## 1.1 Notions of equivalence

We define six notions of equivalence between two programs $P_1$ and $P_2$. The third notion refers to reactive programs, whereas the others to transformational programs.

1. **Partial equivalence:** Given the same inputs, any two terminating executions of $P_1$ and $P_2$ return the same value.
2. **Mutual termination:** Given the same inputs, $P_1$ terminates if and only if $P_2$ terminates.

[1] Without going into the technical details, let us mention that there are various abstraction and decomposition opportunities that are only relevant when proving equivalence. The same observation is well known in the hardware domain, where equivalence checking of circuits is considered computationally easier in practice than model-checking.

[2] CBMC, developed by D. Kroening, allows the user to define a bound $k_i$ on the number of iterations that each loop $i$ in a given ANSI-C program is taken. This enables CBMC to symbolically characterize the full set of possible executions restricted by these bounds, by a decidable formula $f$. The existence of a solution to $f \wedge \neg a$, where $a$ is a user defined assertion, implies the existence of a path in the program that violates $a$. Otherwise, we say that CBMC established the $K$-correctness of the checked assertions, where $K$ denotes the sequence of loop bounds. By default $f$ and $a$ are reduced to propositional formulas.

3. **Reactive equivalence:** Given the same inputs, $P_1$ and $P_2$ emit the same output sequence.
4. $k$**-equivalence:** Given the same inputs, every two executions of $P_1$ and $P_2$ where
   - each loop iterates up to $k$ times, and
   - each recursive call is not deeper than $k$,

   generate the same output.
5. **Total equivalence:** The two programs are partially equivalent and both terminate.
6. **Full equivalence:** The two programs are partially equivalent and mutually terminate.

Comments on this list:

– Only the fourth notion of equivalence in this list is decidable, assuming the program variables range over finite domains.
– The third notion is targeted at reactive programs, although it is relevant to terminating programs as well (in fact it generalizes the first two notions of equivalence). It assumes that inputs are read and outputs are written during the execution of the program.
– The fifth notion of equivalence resembles that of Bouge and Cachera's [2].
– The definitions of "strong equivalence" and "functional equivalence" in [9] and [13], respectively, are almost equivalent to our definition of full equivalence, with the difference that they also require that the two programs have the same set of variables.

## 1.2 The three rules that we prove

The three rules that we prove in this work correspond to the first three notions of equivalence. The rules are not simple to describe without the proper notation. We will give a sketch of these rules here nevertheless. In all rules we begin with a one-to-one mapping between the procedures on both sides such that mapped procedures have the same prototype.[3] If no such mapping is possible, it may be possible to reach such a mapping through inlining, and if this is impossible then our rules are not applicable, at least not for proving the equivalence of full programs.

1. The first rule, called (PROC- P- EQ), can help proving partial equivalence. The rule is based on the following observation. Let $F$ and $G$ be two procedures mapped to one another. Assume that all the mapped procedure calls in $F$ and $G$ return the same values for equivalent arguments. Now suppose that this assumption allows us to prove that $F$ and $G$ are partially equivalent. If these assumptions are correct for every pair of mapped procedures, then we can conclude that all mapped procedures are partially equivalent.
2. The second rule, called (M- TERM), can help proving mutual termination. The rule is based on the following observation. If all paired procedures satisfy:
   - Computational equivalence (e.g. prove by Rule 1), and
   - the conditions under which they call each pair of mapped procedures are equal, and
   - the read arguments of the called procedures are the same when they are called

   then all paired procedures mutually terminate.
3. The third rule, called (REACT- EQ), can help proving that every two mapped procedures are reactive-equivalent. Let $F$ and $G$ be such a mapped pair of procedures. Reactive equivalence means that in every two subcomputations through $F$ and $G$ that are input

[3] We refer to procedures rather than functions from hereon. The *prototype* of a procedure is the sequence of types of the procedure's read and write arguments. In the context of LPL, the programming language that we define below, there is only one type and hence prototypes can be characterized by the number of arguments.

equivalent (this means that they read the same sequence of inputs and are called with the same arguments), the sequence of outputs is the same as well.
If all paired procedures satisfy:

- given the same arguments and the same input sequences, they return the same values (this is similar to the first rule, the difference being that here we also consider the inputs consumed by the procedure during its execution), and
- they consume the same number of inputs, and
- the interleaved sequence of procedure calls and values of output statements inside the mapped procedures is the same (and the procedure calls are made with the same arguments),

then all mapped procedures are reactive equivalent.

Checking all three rules can be automated.

The description of the rules and their proof of soundness refer to a simple programming language called Linear Procedure Language (LPL), which we define in Sect. 2.1, together with its operational semantics. In Sects. 3–5 we state the three inference rules respectively and prove their soundness. Each rule is accompanied with an example.

## 2 Preliminaries

*Notation of sequences.* An $n$-long sequence is denoted by $\langle l_0, \ldots, l_{n-1}\rangle$ or by $\langle l_i\rangle_{i\in\{0,\ldots,n-1\}}$. If the sequence is infinite we write $\langle l_i\rangle_{i\in\{0,\ldots\}}$. Given two sequences $\overline{a} = \langle a_i\rangle_{i\in\{0,\ldots,n-1\}}$ and $\overline{b} = \langle b_i\rangle_{i\in\{0,\ldots,m-1\}}$,

$$\overline{a} \cdot \overline{b}$$

is their concatenation of length $n + m$.

We overload the equality sign (=) to denote sequence equivalence. Given two finite sequences $\overline{a}$ and $\overline{b}$

$$(\overline{a} = \overline{b}) \Leftrightarrow (|\overline{a}| = |\overline{b}| \wedge \forall i \in \{0, \ldots, |\overline{a}| - 1\}.\ a_i = b_i),$$

where $|\overline{a}|$ and $|\overline{b}|$ denote the number of elements in $\overline{a}$ and $\overline{b}$, respectively.

If both $\overline{a}$ and $\overline{b}$ are infinite then

$$(\overline{a} = \overline{b}) \Leftrightarrow (\forall i \geq 0.\ a_i = b_i),$$

and if exactly one of $\{\overline{a}, \overline{b}\}$ is infinite then $\overline{a} \neq \overline{b}$.

*Parentheses and brackets* We use a convention by which arguments of a function are enclosed in parenthesis, as in $f(e)$, when the function maps values within a single domain. If it maps values between different domains we use brackets, as in $f[e]$. References to vector elements, for example, belong to the second group, as they map between indices and values in the domain of the vector. Angled brackets ($\langle\cdot\rangle$) are used for both sequences as shown above, and for tuples.

### 2.1 The programming language

To define the programming language we assume a set of procedure names $Proc = \{p_0, \ldots, p_m\}$, where $p_0$ has a special role as the *root procedure* (the equivalent of "main" in C). Let

$\mathbb{D}$ be a domain that contains the constants TRUE and FALSE, and no subtypes. Let $O_{\mathbb{D}}$ be a set of operations (functions and predicates) over $\mathbb{D}$. We define a set of variables over this domain: $V = \bigcup_{p \in Proc} V_p$, where $V_p$ is the set of variables of a procedure $p$. The sets $V_p$, $p \in Proc$ are pairwise disjoint. For expression $e$ over $\mathbb{D}$ and $V$ we denote by $vars[e]$ the set of variables that appear in $e$.

The LPL language is modeled after PLW [6], but is different in various aspects. For example, it does not contain loops and allows only procedure calls by value-return.

**Definition 1 (Linear Procedure Language (LPL))** The *linear procedure language* (LPL) is defined by the following grammar (lexical elements of LPL are in bold, and $S$ denotes *Statement* constructs):

$$\begin{aligned}
&Program :: \ \langle \textbf{procedure}\ p(\textbf{val}\ \overline{arg\text{-}r_p};\ \textbf{ret}\ \overline{arg\text{-}w_p}){:}S_p\rangle_{p \in Proc}\\
&S :: \ x := e \mid S;S \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi} \mid\\
&\qquad \textbf{call}\ p(\overline{e};\overline{x}) \mid \textbf{return}
\end{aligned}$$

where $p \in Proc$, $e$ is an expression over $O_{\mathbb{D}}$, and $B$ is a predicate over $O_{\mathbb{D}}$. $\overline{arg\text{-}r_p}$, $\overline{arg\text{-}w_p}$ are vectors of $V_p$ variables called, respectively, *read formal arguments* and *write formal arguments*, and are used in the *body* $S_p$ of the procedure named $p$. In a procedure call "**call** $p(\overline{e};\overline{x})$", the expressions $\overline{e}$ are called the *actual input arguments* and $\overline{x}$ are called the *actual output variables*. The following constraints are assumed:

1. The only variables that can appear in the *procedure body* $S_p$ are from $V_p$.
2. For each procedure call "**call** $p(\overline{e},\overline{x})$" the lengths of $\overline{e}$ and $\overline{x}$ are equal to the lengths of $\overline{arg\text{-}r_p}$ and $\overline{arg\text{-}w_p}$, respectively.
3. **return** must appear at the end of any procedure body $S_p$ ($p \in Proc$). □

For simplicity LPL is defined so it does not permit global variables and iterative expressions like **while** loops. Both of these syntactic restrictions do not constrain the expressive power of the language: global variables can be passed as part of the list of arguments of each procedure, and loops can be rewritten as recursive expressions.

**Definition 2 (An LPL augmented by location labels)** An LPL program augmented with location labels is derived from an LPL program $P$ by adding unique labels **before**$[S]$ and **after**$[S]$ for each statement $S$, right before and right after $S$, respectively. As an exception, for two composed statements $S_1$ and $S_2$ (i.e., $S_1$; $S_2$), we do not dedicate a label for after$[S_1]$; rather, we define after$[S_1]$ = before$[S_2]$. □

*Example 2* Consider the LPL program $P$ at the left of Fig. 2, defined over the domain $\mathbb{Z} \cup \{$TRUE, FALSE$\}$ for which, among others, the operations $+, -, =$ are well-defined. The same program augmented with location labels appears on the right of the same figure. □

```
procedure p1(val x; ret y):
    z := x + 1;
    y := x − 1;
    return
procedure p0(val w; ret w):
    if (w = 0) then
        w := 1
    else w := 2
    fi;
    call p1(w;w);
    return
```

```
procedure p1(val x; ret y):
    l1  z := x + 1;
    l2  y := x − 1;
    l3  return  l4
procedure p0(val w; ret w):
        l5  if (w = 0) then
            l6  w := 1  l7
            else l8  w := 2  l9
        fi;
        l10  call p1(w;w);  l11
        return  l12
```

**Fig. 2** An LPL program (*left*) and its augmented version (*right*)

The partial order $\prec$ of the locations is any order which satisfies :

1. For any statement $S$, before[$S$] $\prec$ after[$S$].
2. For an **if** statement $S$ : **if** $B$ **then** $S_1$ **else** $S_2$ **fi**, before[$S$] $\prec$ before[$S_1$], before[$S$] $\prec$ before[$S_2$], after[$S_1$] $\prec$ after[$S$] and after[$S_2$] $\prec$ after[$S$].

We denote the set of location labels in the body of procedure $p \in Proc$ by $PC_p$. Together the set of all location labels is $PC \doteq \bigcup_{p \in Proc} PC_p$.

## 2.2 Operational semantics

A computation of a program $P$ in LPL is a sequence of configurations. Each configuration $C = \langle d, O, \overline{pc}, \sigma \rangle$ contains the following elements:

1. The natural number $d$ is the depth of the stack at this configuration.
2. The function $O : \{0, \ldots, d\} \mapsto Proc$ is the *order of procedures* in the stack at this configuration.
3. $\overline{pc} = \langle pc_0, pc_1 \ldots, pc_d \rangle$ is a vector of program location labels[4] such that $pc_0 \in PC_0$ and for each call level $i \in \{1, \ldots, d\}$ $pc_i \in PC_{O[i]}$ (i.e., $pc_i$ "points" into the procedure body that is at the $i$th place in the stack).
4. The function $\sigma : \{0, \ldots, d\} \times V \mapsto \mathbb{D} \cup \{nil\}$ is a *valuation* of the variables $V$ of program $P$ at this configuration. The value of variables which are not active at the $i$-th call level is invalid i.e., for $i \in \{0, \ldots, d\}$, if $O[i] = p$ and $v \in V \setminus V_p$ then $\sigma[\langle i, v \rangle] = nil$ where $nil \notin \mathbb{D}$ denotes an invalid value.

A valuation is implicitly defined over a configuration. For an expression $e$ over $\mathbb{D}$ and $V$, we define the value of $e$ in $\sigma$ in the natural way, i.e., each variable evaluates according to the procedure and the stack depth defined by the configuration. More formally, for a configuration $C = \langle d, O, \overline{pc}, \sigma \rangle$ and a variable $x$:

$$\sigma[x] \doteq \begin{cases} \sigma[\langle d, x \rangle] & if \;\; x \in V_p \;\; and \;\; p = O[d] \\ nil & otherwise \end{cases}$$

This definition extends naturally to a vector of expressions.

When referring to a specific configuration $C$, we denote its elements $d, O, \overline{pc}, \sigma$ with $C.d, C.O, C.\overline{pc}, C.\sigma[x]$, respectively.

For a valuation $\sigma$, expression $e$ over $\mathbb{D}$ and $V$, levels $i, j \in \{0, \ldots, d\}$, and a variable $x$, we denote by $\sigma[\langle i, e \rangle | \langle j, x \rangle]$ a valuation identical to $\sigma$ other than the valuation of $x$ at level $j$, which is replaced with the valuation of $e$ at level $i$. When the respective levels are clear from the context, we may omit them from the notation.

Finally, we denote by $\sigma|_i$ a valuation $\sigma$ *restricted to level* $i$, i.e., $\sigma|_i[v] \doteq \sigma[\langle i, v \rangle]$ $(v \in V)$.

For a configuration $C = \langle d, O, \overline{pc}, \sigma \rangle$ we denote by current-label[$C$] the program location label at the procedure that is topmost on the stack, i.e., current-label[$C$] $\doteq pc_d$.

**Definition 3 (Initial and Terminal configurations in LPL)** A configuration $C = \langle d, O, \overline{pc}, \sigma \rangle$ with current-label[$C$] = before[$S_{p_0}$] is called the *initial* configuration and must satisfy $d = 0$ and $O[0] = p_0$. A configuration with current-label[$C$] = after[$S_{p_0}$] is called the *terminal* configuration. □

**Definition 4 (Transition relation in LPL)** Let "$\rightarrow$" be the least relation among configurations which satisfies: if $C \rightarrow C'$, $C = \langle d, O, \overline{pc}, \sigma \rangle$, $C' = \langle d', O', \overline{pc}', \sigma' \rangle$ then:

[4] $\overline{pc}$ can be thought of as a stack of program counters, hence the notation.

1. If current-label$[C]$ = before$[S]$ for some assign construct $S =$ "$x := e$" then $d' = d$, $O' = O$, $\overline{pc}' = \langle pc_i \rangle_{i \in \{0,\ldots,d-1\}} \cdot \langle \text{after}[S] \rangle$, $\sigma' = \sigma[e|x]$.
2. If current-label$[C]$ = before$[S]$ for some construct

$$S = \text{"}\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\text{"}$$

then

$$d' = d,\ O' = O,\ \overline{pc}' = \langle pc_i \rangle_{i \in \{0,\ldots,d-1\}} \cdot \langle lab_B \rangle,\ \sigma' = \sigma$$

where

$$lab_B = \begin{cases} \text{before}[S_1] & \text{if } \sigma[B] = \text{TRUE} \\ \text{before}[S_2] & \text{if } \sigma[B] = \text{FALSE} \end{cases}$$

3. If current-label$[C]$ = after$[S_1]$ or current-label$[C]$ = after$[S_2]$ for some construct

$$S = \text{"}\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\text{"}$$

then

$$d' = d,\ O' = O,\ \overline{pc}' = \langle pc_i \rangle_{i \in \{0,\ldots,d-1\}} \cdot \langle \text{after}[S] \rangle,\ \sigma' = \sigma$$

4. If current-label$[C]$ = before$[S]$ for some call construct $S =$ "**call** $p(\overline{e}; \overline{x})$" then $d' = d + 1$, $O' = O \cdot \langle p \rangle$, $\overline{pc}' = \langle pc_i \rangle_{i \in \{0,\ldots,d-1\}} \cdot \langle \text{after}[S] \rangle \cdot \langle \text{before}[S_p] \rangle$, $\sigma' = \sigma[\langle d, e_1 \rangle | \langle d+1, (arg\text{-}r_p)_1 \rangle] \ldots [\langle d, e_l \rangle | \langle d+1, (arg\text{-}r_p)_l \rangle]$ where $\overline{arg\text{-}r_p}$ is the vector of formal read variables of procedure $p$ and $l$ is its length.
5. If current-label$[C]$ = before$[S]$ for some return construct $S =$ "**return**" and $d > 0$ then $d' = d - 1$, $O' = \langle O_i \rangle_{i \in \{1,\ldots,d-1\}}$, $\overline{pc}' = \langle pc_i \rangle_{i \in \{0,\ldots,d-1\}}$, $\sigma' = \sigma[\langle d, (arg\text{-}w_p)_1 \rangle | \langle d-1, x_1 \rangle] \ldots [\langle d, (arg\text{-}w_p)_l \rangle | \langle d-1, x_l \rangle]$ where $\overline{arg\text{-}w_p}$ is the vector of formal write variables of procedure $p$, $l$ is its length, and $\overline{x}$ are the actual output variables of the **call** statement immediately before $pc_{d-1}$.
6. If current-label$[C]$ = before$[S]$ for some return construct $S =$ "**return**" and $d = 0$ then $d' = 0$, $O' = \langle p_0 \rangle$, $\overline{pc}' = \langle \text{after}[S_{p_0}] \rangle$ and $\sigma' = \sigma$. □

Note that the case of current-label$[C]$ = before$[S]$ for a construct $S = S_1; S_2$ is always covered by one of the cases in the above definition.

Another thing to note is that all write arguments are copied to the actual variables following a **return** statement. This solves possible problems that may occur if the same variable appears twice in the list of write arguments.

## 2.3 Computations and subcomputations of LPL programs

A computation of a program $P$ in LPL is a sequence of configurations $\overline{C} = \langle C_0, C_1, \ldots \rangle$ such that $C_0$ is an initial configuration and for each $i \leq |\overline{C}| - 1$ we have $C_i \to C_{i+1}$. If the computation is finite then the last configuration must be terminal.

The proofs throughout this article will be based on the notion of subcomputations. We distinguish between several types of subcomputations, as follows (an example will be given after the definitions):

**Definition 5 (Subcomputation *at* a level)** A continuous subsequence of a computation is a *subcomputation at level d* if all its configurations have the same stack depth $d$. □

Clearly every subcomputation at a level is finite.

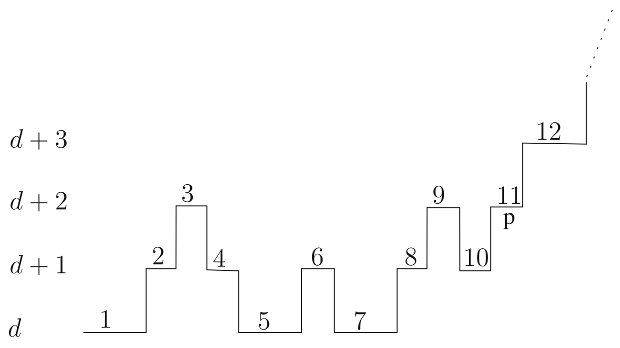

**Fig. 3** A computation through various stack levels. Each rise corresponds to a procedure call, and each fall to a **return** statement

**Definition 6 (Maximal subcomputation *at* a level)** A *maximal* subcomputation at level $d$ is a subcomputation at level $d$, such that the successor of its last configuration has stack-depth different than $d$, or $d = 0$ and its last configuration is equal to $after[S_0]$. □

**Definition 7 (Subcomputation *from* a level)** A continuous subsequence of a computation is a *subcomputation from level d* if its first configuration $C_0$ has stack depth $d$, current-label $[C_0] =$ before$[S_p]$ for some procedure $p$ and all its configurations have a stack depth of at least $d$. □

**Definition 8 (Maximal subcomputation *from* a level)** A *maximal* subcomputation from level $d$ is a subcomputation from level $d$ which is either

- infinite, or
- finite, and,
  - if $d > 0$ the successor of its last configuration has stack-depth smaller than $d$, and
  - if $d = 0$, then its last configuration is equal to $after[S_0]$. □

A finite maximal subcomputation is also called *closed*.

*Example 3* In Fig. 3, each whole segment corresponds to a maximal subcomputation *at* its respective stack level, e.g., segment 2 is a maximal subcomputation at level $d + 1$, the subsequence 8–11 is a finite (but not maximal) subcomputation *from* level $d + 1$, and the subsequence 2–4 is a maximal subcomputation from level $d + 1$.

Let $\pi$ be a computation and $\pi'$ a continuous subcomputation of $\pi$. We will use the following notation to refer to different configurations in $\pi'$:

- **first**$[\pi']$ denotes the first configuration in $\pi'$.
- **last**$[\pi']$ denotes the last configuration in $\pi'$, in case $\pi'$ is finite.
- **pred**$[\pi']$ is the configuration in $\pi$ for which pred$[\pi'] \to$ first$[\pi']$.
- **succ**$[\pi']$ is the configuration in $\pi$ such that last$[\pi'] \to$ succ$[\pi']$.

## 2.4 An assumption about the programs we compare

Two procedures

$$\textbf{procedure } F(\textbf{val } \overline{arg\text{-}r_F}; \textbf{ ret } \overline{arg\text{-}w_F}),$$
$$\textbf{procedure } G(\textbf{val } \overline{arg\text{-}r_G}; \textbf{ ret } \overline{arg\text{-}w_G})$$

are said to have an equivalent prototype if $|\overline{arg\text{-}r_F}| = |\overline{arg\text{-}r_G}|$ and $|\overline{arg\text{-}w_F}| = |\overline{arg\text{-}w_G}|$.

We will assume that the two LPL programs $P_1$ and $P_2$ that we compare have the following property: $|Proc[P_1]| = |Proc[P_2]|$, and there is a 1-1 and onto mapping $map : Proc[P_1] \mapsto Proc[P_2]$ such that if $\langle F, G\rangle \in map$ then $F$ and $G$ have an equivalent prototype.

Programs that we wish to prove equivalent and do not fulfill this requirement, can sometimes be brought to this state by applying inlining of procedures that can not be mapped.

## 3 A proof rule for partial procedure equivalence

Given the operational semantics of LPL, we now proceed to define a proof rule for the partial equivalence of two LPL procedures. The rule refers to finite computations only. We delay the discussion on more general cases to Sects. 4 and 5.

Our running example for this section will be the two programs in Fig. 4, which compute recursively yet in different ways the GCD (Greatest Common Divisor) of two positive integers. We would like to prove that when they are called with the same inputs, they return the same result.

### 3.1 Definitions

We now define various terms and notations regarding subcomputations through procedure bodies. All of these terms refer to subcomputations that begin right before the first statement in the procedure and end just before the **return** statement (of the same procedure at the same level), and use the formal arguments of the procedure. We will overload these terms, however, when referring to subcomputations that begin right before the **call** statement to the same procedure and end right after it, and consequently use the actual arguments of the procedure. This overloading will repeat itself in future sections as well.

**Definition 9 (Argument-equivalence of subcomputations with respect to procedures)** Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map$, for any two computations $\pi_1$ in $P_1$ and $\pi_2$ in $P_2$, $\pi_1'$ and $\pi_2'$ are *argument-equivalent with respect to $F$ and $G$* if the following holds:

1. $\pi_1'$ and $\pi_2'$ are maximal subcomputations of $\pi_1$ and $\pi_2$ from some levels $d_1$ and $d_2$, respectively,
2. current-label[first[$\pi_1'$]] = before[$F$] and current-label[first[$\pi_2'$]] = before[$G$], and
3. first[$\pi_1'$].$\sigma[\overline{arg\text{-}r_F}]$ = first[$\pi_2'$].$\sigma[\overline{arg\text{-}r_G}]$, □

**Definition 10 (Partial computational equivalence of procedures)** If for every argument-equivalent finite subcomputations $\pi_1'$ and $\pi_2'$ (these are closed by definition) with respect to two procedures $F$ and $G$,

$$\text{last}[\pi_1'].\sigma[\overline{arg\text{-}w_F}] = \text{last}[\pi_2'].\sigma[\overline{arg\text{-}w_G}]$$

then $F$ and $G$ are *partially computationally equivalent.* □

```
procedure gcd_1(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        l_1  call gcd_1(b, a; g)  l_3
    fi;
    return
```

```
procedure gcd_2(val x,y; ret z):
    z := x;
    if y > 0 then
        l_2  call gcd_2(y, z mod y; z)  l_4
    fi;
    return
```

**Fig. 4** Two procedures to calculate GCD of two positive integers. For better readability we only show the labels that we later refer to

Denote by $comp\text{-}equiv(F, G)$ the fact that $F$ and $G$ are partially computationally equivalent. The computational equivalence is only partial because it does not consider infinite computations. From hereon when we talk about computational equivalence we mean partial computational equivalence.

Our proof rule uses *uninterpreted procedures*, which are useful for reasoning about an abstract system. The only information that the decision procedure has about them is that they are consistent, i.e., that given the same inputs, they produce the same outputs. We still need a semantics for such procedures, in order to be able to define subcomputations that go through them. In terms of the semantics, then, an uninterpreted procedure $U$ is the same as an empty procedure in LPL (a procedure with a single statement—**return**), other than the fact that it preserves the **congruence condition**: For every two subcomputations $\pi_1$ and $\pi_2$ through $U$,

$$\begin{array}{l}\text{first}[\pi_1].\sigma[\,\overline{arg\text{-}r_U}\,] = \text{first}[\pi_2].\sigma[\,\overline{arg\text{-}r_U}\,]\\ \rightarrow\\ \text{last}[\pi_1].\sigma[\,\overline{arg\text{-}w_U}\,] = \text{last}[\pi_2].\sigma[\,\overline{arg\text{-}w_U}\,].\end{array} \tag{1}$$

There are well known decision procedures for reasoning about formulas that involve uninterpreted *functions*—see, for example, Shostak's algorithm [14], and accordingly most theorem provers support them. Such algorithms can be easily adapted to handle procedures rather than functions.

## 3.2 Rule (PROC- P- EQ)

Defining the proof rule requires one more definition.

Let UP be a mapping of the procedures in $Proc[P_1] \cup Proc[P_2]$ to respective uninterpreted procedures, such that:

$$\langle F, G\rangle \in map \iff \text{UP}(F) = \text{UP}(G), \tag{2}$$

and such that each procedure is mapped to an uninterpreted procedure with an equivalent prototype.

**Definition 11 (Isolated procedure)** The *isolated* version of a procedure $F$, denoted $F^{UP}$, is derived from $F$ by replacing all of its procedure calls by calls to the corresponding uninterpreted procedures, i.e., $F^{UP} \doteq F[f \leftarrow \text{UP}(f) | f \in Proc[P]]$. □

For example, Fig. 6 presents an isolated version of the programs in Fig. 4.

Rule (PROC- P- EQ), appearing in Fig. 5, is based on the following observation. Let $F$ and $G$ be two procedures such that $\langle F, G\rangle \in map$. If assuming that all the mapped procedure calls in $F$ and $G$ return the same values for equivalent arguments enables us to prove that $F$ and $G$ are equivalent, then we can conclude that $F$ and $G$ are equivalent.

The rule assumes a proof system $\mathbb{L}_{\mathbb{UP}}$. $\mathbb{L}_{\mathbb{UP}}$ is any sound proof system for a restricted version of the programming language in which there are no calls to interpreted procedures,

$$\frac{\begin{array}{ll}(3.1) & \forall\langle F, G\rangle \in map.\ \{\\ (3.2) & \qquad \vdash_{\mathbb{L}_{\mathbb{UP}}}\ comp\text{-}equiv(F^{UP}, G^{UP})\ \ \}\end{array}}{(3.3) \qquad \forall\langle F, G\rangle \in map.\ comp\text{-}equiv(F, G)}\ (\text{PROC-P-EQ}) \tag{3}$$

**Fig. 5** Rule (PROC- P- EQ): An inference rule for proving the partial equivalence of procedures

```
procedure $gcd_1$(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        call H(b, a; g)
    fi;
    return
```

```
procedure $gcd_2$(val x,y; ret z):
    z := x;
    if $y > 0$ then
        call H(y, z mod y; z)
    fi;
    return
```

**Fig. 6** After isolation of the procedures, i.e., replacing their procedure calls with calls to the uninterpreted procedure $H$

and hence, in particular, no recursion[5], and it can reason about uninterpreted procedures. $\mathbb{L}_{\mathbb{UP}}$ is not required to be complete, because (PROC- P- EQ) is incomplete in any case. Nevertheless, completeness is desirable since it makes the rule more useful.

*Example 4* Following are two instantiations of rule (PROC- P- EQ).

– The two programs contain one recursive procedure each, called $f$ and $g$ such that $map = \{\langle f, g\rangle\}$.

$$\frac{\vdash_{\mathbb{L}_{\mathbb{UP}}} \; comp\text{-}equiv(f[f \leftarrow \mathrm{UP}(f)], g[g \leftarrow \mathrm{UP}(g)])}{comp\text{-}equiv(f, g)}$$

Recall that $f[f \leftarrow \mathrm{UP}(f)]$ means that the call to $f$ inside $f$ is replaced with a call to UP(f) (isolation).

– The two compared programs contain two mutually recursive procedures each, $f_1$, $f_2$ and $g_1$, $g_2$, respectively, such that $map = \{\langle f_1, g_1\rangle, \langle f_2, g_2\rangle\}$, and $f_1$ calls $f_2$, $f_2$ calls $f_1$, $g_1$ calls $g_2$ and $g_2$ calls $g_1$.

$$\frac{\begin{array}{c}\vdash_{\mathbb{L}_{\mathbb{UP}}} comp\text{-}equiv(f_1[f_2 \leftarrow \mathrm{UP}(f_2)], g_1[g_2 \leftarrow \mathrm{UP}(g_2)]),\\ \vdash_{\mathbb{L}_{\mathbb{UP}}} comp\text{-}equiv(f_2[f_1 \leftarrow \mathrm{UP}(f_1)], g_2[g_1 \leftarrow \mathrm{UP}(g_1)])\end{array}}{comp\text{-}equiv(f_1, g_1),\; comp\text{-}equiv(f_2, g_2)}$$

□

*Example 5* Consider once again the two programs in Fig. 4. There is only one procedure in each program, which we naturally map to one another. Let $H$ be the uninterpreted procedure to which we map $gcd_1$ and $gcd_2$, i.e., $H = \mathrm{UP}(gcd_1) = \mathrm{UP}(gcd_2)$. Figure 6 presents the isolated programs.

To prove the computational equivalence of the two procedures, we need to first translate them to formulas expressing their respective transition relations. A convenient way to do so is to use Static Single Assignment (SSA) [3]. Briefly, this means that in each assignment of the form `x = exp;` the left-hand side variable `x` is replaced with a new variable, say $\mathtt{x}_1$. Any reference to `x` after this line and before `x` is potentially assigned again, is replaced with the new variable $\mathtt{x}_1$ (recall that this is done in a context of a program without loops). In addition, assignments are guarded according to the control flow. After this transformation, the statements are conjoined: the resulting equation represents the states of the original program. If a subcomputation through a procedure is valid then it can be associated with an assignment that satisfies the SSA form of this procedure.

[5] In LPL there are no loops, but in case (PROC- P- EQ) is applied to other languages, $\mathbb{L}_{\mathbb{UP}}$ is required to handle a restricted version of the language with no procedure calls, recursion or loops. Indeed, under this restriction there are sound and complete decision procedures for deciding the validity of assertions over popular programming languages such as C, as was mentioned in the introduction.

The SSA form of $gcd_1$ is

$$T_{gcd_1} = \begin{pmatrix} a_0 = a & \wedge \\ b_0 = b & \wedge \\ b_0 = 0 \rightarrow g_0 = a_0 & \wedge \\ (b_0 \neq 0 \rightarrow a_1 = (a_0 \mod b_0)) \wedge (b_0 = 0 \rightarrow a_1 = a_0) & \wedge \\ (b_0 \neq 0 \rightarrow H(b_0, a_1; g_1)) \wedge (b_0 = 0 \rightarrow g_1 = g_0) & \wedge \\ g = g_1 & \end{pmatrix}. \quad (4)$$

The SSA form of $gcd_2$ is

$$T_{gcd_2} = \begin{pmatrix} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ (y_0 > 0 \rightarrow H(y_0, (z_0 \mod y_0); z_1)) \wedge (y_0 \leq 0 \rightarrow z_1 = z_0) & \wedge \\ z = z_1 & \end{pmatrix}. \quad (5)$$

The premise of rule (PROC- P- EQ) requires proving computational equivalence (see Definition 10), which in this case amounts to proving the validity of the following formula over positive integers:

$$(a = x \wedge b = y \wedge T_{gcd_1} \wedge T_{gcd_2}) \quad \rightarrow \quad g = z. \quad (6)$$

Many theorem provers can prove such formulas fully automatically, and hence establish the partial computational equivalence of $gcd_1$ and $gcd_2$. □

It is important to note that while the premise refers to procedures that are isolated from other procedures, the consequent refers to the original procedures. Hence, while $\mathbb{L}_{\mathbb{UP}}$ is required to reason about executions of bounded length (the length of one procedure body) the consequent refers to unbounded executions.

To conclude this section, let us mention that rule (PROC- P- EQ) is inspired by Hoare's rule for recursive procedures:

$$\frac{\{p\}\textbf{call}\ proc\{q\} \vdash_H \{p\}S\{q\}}{\{p\}\textbf{call}\ proc\{q\}} \quad \text{(REC)}$$

(where $S$ is the body of procedure $proc$). Indeed, both in this rule and in rule (PROC- P- EQ), the premise requires to prove that the body of the procedure without its recursive calls satisfies the pre-post condition relation that we wish to establish, assuming the recursive calls already do so.

## 3.3 Rule (PROC- P- EQ) is sound

Let $\pi$ be a computation of some program $P_1$. Each subcomputation $\pi'$ *from* level $d$ consists of a set of maximal subcomputations *at* level $d$, which are denoted by $in(\pi', d)$, and a set of maximal subcomputations *from* level $d+1$, which are denoted by $from(\pi', d+1)$. The members of these two sets of computations alternate in $\pi'$. For example, in the left drawing in Fig. 7, segments 2, 4, 8 are separate subcomputations at level $d+1$, and segments 3, and 5–7 are subcomputations from level $d+2$.

**Definition 12 (Stack-level tree)** A *stack-level tree* of a maximal subcomputation $\pi$ from some level, is a tree in which each node at height $d$ ($d > 0$) represents the set of subcomputations *at* level $d$ from the time the computation entered level $d$ until it returned to its

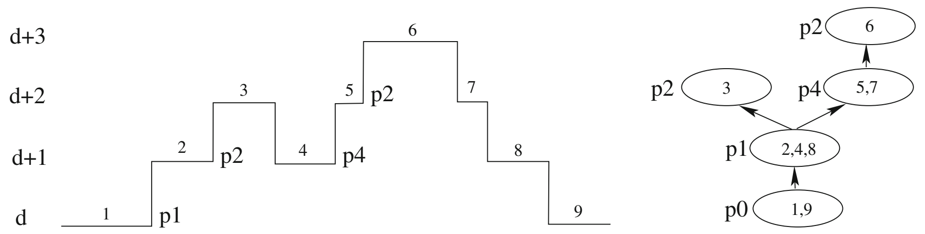

**Fig. 7** A computation and its stack levels (*left*). The numbering on the horizontal segments are for reference only. The stack-level tree corresponding to the computation on the *left* (*right*)

calling procedure at level $d-1$. Node $n'$ is a child of a node $n$ if and only if it contains one subcomputation that is a continuation (in $\pi$) of a subcomputation in $n$. □

Note that the root of a stack-level tree is the node that contains first$[\pi]$ in one of its subcomputations. The leafs are closed subcomputations from some level which return without executing a procedure call. Also note that the subcomputations in a node at level $d$ are all part of the same closed subcomputation $\pi'$ *from* level $d$ (this is exactly the set $in(\pi', d)$).

The *stack-level tree depth* is the maximal length of a path from its root to some leaf. This is also the maximal difference between the depth of the level of any of its leafs and the level of its root. If the stack-level tree is not finite then its depth is undefined.

Denote by $d[n]$ the level of node $n$ and by $p[n]$ the procedure associated with this node.

*Example 6* Figure 7 demonstrates a subcomputation $\pi$ from level $d$ (left) and its corresponding stack-level tree (upside down, in order to emphasize its correspondence to the computation). The set $in(\pi, d) = \{1, 9\}$ is represented by the root. Each rise in the stack level is associated with a procedure call (in this case, calls to p1, p2, p4, p2), and each fall with a **return** statement. To the left of each node $n$ in the tree, appears the procedure $p[n]$ (here we assumed that the computation entered level $d$ due to a call to a procedure p0). The depth of this stack-level tree is 4. □

**Theorem 1** **(Soundness)** *If the proof system* $\mathbb{L}_{\mathbb{UP}}$ *is sound then the rule* (PROC- P- EQ) *is sound.*

*Proof* By induction on the depth $d$ of the stack-level tree. Since we consider only finite computations, the stack-level trees are finite and their depths are well defined. Let $P_1$ and $P_2$ be two programs in LPL, $\pi_1$ and $\pi_2$ closed subcomputations from some levels in $P_1$ and $P_2$, respectively, $t_1$ and $t_2$ the stack-level trees of these computations, $n_1$ and $n_2$ the root nodes of $t_1$ and $t_2$, respectively. Also, let $F = p[n_1]$ and $G = p[n_2]$ where $\langle F, G\rangle \in map$. Assume also that $\pi_1$ and $\pi_2$ are argument-equivalent with respect to $F$ and $G$.

*Base* If both $n_1$ and $n_2$ are leafs in $t_1$ and $t_2$ then the conclusion is proven by the premise of the rule without using the uninterpreted procedures. As $\pi_1$ and $\pi_2$ contain no calls to procedures, then they are also valid computations through $F^{UP}$ and $G^{UP}$, respectively. Therefore, by the soundness of the proof system $\mathbb{L}_{\mathbb{UP}}$, $\pi_1$ and $\pi_2$ must satisfy $comp\text{-}equiv(F^{UP}, G^{UP})$ which entails the equality of $\overline{arg\text{-}w}$ values at their ends. Therefore, $\pi_1$ and $\pi_2$ satisfy the condition in $comp\text{-}equiv(F, G)$.

*Step* Assume the consequent (3.3) is true for all stack-level trees of depth at most $i$. We prove the consequent for computations with stack-level trees $t_1$ and $t_2$ such that at least one of them is of depth $i+1$.

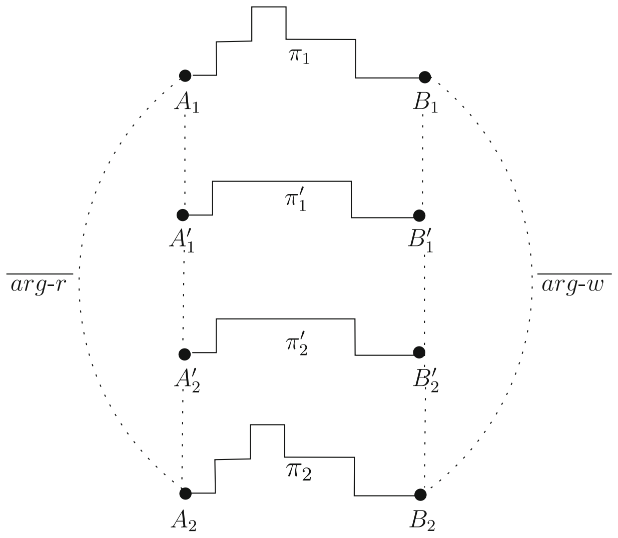

**Fig. 8** A diagram for the proof of Theorem 1. Dotted lines indicate an equivalence (either that we assume as a premise or that we need to prove) in the argument that labels the line. We do not write all labels to avoid congestion—see more details in the proof. $\pi_1$ is a subcomputation through $F$. $\pi'_1$ is the corresponding subcomputation through $F^{UP}$, the isolated version of $F$. The same applies to $\pi_2$ and $\pi'_2$ with respect to $G$. The induction step shows that if the read arguments are the same in $A.1$ and $A.2$, then the write arguments have equal values in $B.1$ and $B.2$

1. Consider the computation $\pi_1$. We construct a computation $\pi'_1$ in $F^{UP}$, which is the same as $\pi_1$ in the level of $n_1$, with the following change. Each subcomputation of $\pi_1$ in a deeper level caused by a call $c_F$, is replaced by a subcomputation through an uninterpreted procedure UP($callee[c_F]$), which returns the same value as returned by $c_F$ (where $callee[c_F]$ is the procedure called in the **call** statement $c_F$). In a similar way we construct a computation $\pi'_2$ in $G^{UP}$ corresponding to $\pi_2$.
   The notations we use in this proof correspond to Fig. 8. Specifically,
   $$A_1 = \text{first}[\pi_1],\ A'_1 = \text{first}[\pi'_1],\ A'_2 = \text{first}[\pi'_2],\ A_2 = \text{first}[\pi_2],$$
   $$B_1 = \text{last}[\pi_1],\ B'_1 = \text{last}[\pi'_1], B'_2 = \text{last}[\pi'_2],\ B_2 = \text{last}[\pi_2].$$
2. As $\pi_1$ and $\pi_2$ are argument-equivalent we have
   $$A_1.\sigma[\,\overline{arg\text{-}r}_F\,] = A_2.\sigma[\,\overline{arg\text{-}r}_G\,].$$
   By definition,
   $$A'_1.\sigma[\,\overline{arg\text{-}r}_F\,] = A_1.\sigma[\,\overline{arg\text{-}r}_F\,]$$
   and
   $$A'_2.\sigma[\,\overline{arg\text{-}r}_G\,] = A_2.\sigma[\,\overline{arg\text{-}r}_G\,].$$
   By transitivity of equality $A'_1.\sigma[\,\overline{arg\text{-}r}_F\,] = A'_2.\sigma[\,\overline{arg\text{-}r}_G\,]$.
3. We now prove that the subcomputations $\pi'_1$ and $\pi'_2$ are valid computations through $F^{UP}$ and $G^{UP}$. As $\pi'_1$ and $\pi'_2$ differ from $\pi_1$ and $\pi_2$ only by subcomputations through uninterpreted procedures (that replace calls to other procedures), we need to check that they

satisfy the congruence condition, as stated in (1). Other parts of $\pi'_1$ and $\pi'_2$ are valid because $\pi_1$ and $\pi_2$ are valid subcomputations. Consider any pair of calls $c_1$ and $c_2$ in $\pi_1$ and $\pi_2$ from the current levels $d[n_1]$ and $d[n_2]$ to procedures $p_1$ and $p_2$, respectively, such that $\langle p_1, p_2\rangle \in map$. Let $c'_1$ and $c'_2$ be the calls to UP($p_1$) and UP($p_2$) which replace $c_1$ and $c_2$ in $\pi'_1$ and $\pi'_2$. Note that UP($p_1$) = UP($p_2$) since $\langle p_1, p_2\rangle \in map$.
By the induction hypothesis, procedures $p_1, p_2$ satisfy $comp\text{-}equiv(p_1, p_2)$ for all subcomputations of depth $\leq i$, and in particular for subcomputations of $\pi_1, \pi_2$ that begin in $c_1$ and $c_2$. By construction, the input and output values of $c_1$ are equal to those of $c'_1$. Similarly, the input and output values of $c_2$ are equal to those of $c'_2$. Consequently, the pair of calls $c'_1$ and $c'_2$ to the uninterpreted procedure UP($p_1$) satisfy the congruence condition. Hence, $\pi'_1$ and $\pi'_2$ are legal subcomputations through $F^{UP}$ and $G^{UP}$.

4. By the rule premise, any two computations through $F^{UP}$ and $G^{UP}$ satisfy $comp\text{-}equiv$ $(F^{UP}, G^{UP})$. Particularly, as $\pi'_1$ and $\pi'_2$ are argument-equivalent by step 2, this entails that $B'_1.\sigma[\overline{arg\text{-}w_F}] = B'_2.\sigma[\overline{arg\text{-}w_G}]$. By construction, $B_1.\sigma[\overline{arg\text{-}w_F}] = B'_1.\sigma[\overline{arg\text{-}w_F}]$ and $B_2.\sigma[\overline{arg\text{-}w_G}] = B'_2.\sigma[\overline{arg\text{-}w_G}]$. Therefore, by transitivity,

$$B_1.\sigma[\,\overline{arg\text{-}w_F}\,] = B_2.\sigma[\,\overline{arg\text{-}w_G}\,],$$

which proves that $\pi_1$ and $\pi_2$ satisfy $comp\text{-}equiv(F, G)$. □

## 4 A proof rule for mutual termination of procedures

Rule (PROC- P- EQ) only proves partial equivalence, because it only refers to finite computations. It is desirable, in the context of equivalence checking, to prove that the two procedures mutually terminate. If, in addition, termination of one of the programs is proven, then "total equivalence" is established.

### 4.1 Definitions

**Definition 13 (Mutual termination of procedures)** If for every pair of argument-equivalent subcomputations $\pi'_1$ and $\pi'_2$ with respect to two procedures $F$ and $G$, it holds that $\pi'_1$ is finite if and only if $\pi'_2$ is finite, then $F$ and $G$ are *mutually terminating*. □

Denote by $mutual\text{-}terminate(F, G)$ the fact that $F$ and $G$ are mutually terminating.

**Definition 14 (Reach equivalence of procedures)** Procedures $F$ and $G$ are *reach-equivalent* if for every pair of argument-equivalent subcomputation $\pi$ and $\tau$ through $F$ and $G$, respectively, for every **call** statement $c_F$ = "**call** $p_1$" in $F$ (in $G$), there exists a call $c_G$ = "**call** $p_2$" in $G$ (in $F$) such that $\langle p_1, p_2\rangle \in map$, and $\pi$ and $\tau$ reach $c_F$ and $c_G$, respectively with the same read arguments, or do not reach them at all.

Denote by $reach\text{-}equiv(F, G)$ the fact that $F$ and $G$ are reach-equivalent. Note that checking for reach-equivalence amounts to proving the equivalence of the "guards" leading to each of the mapped procedure calls (i.e., the conjunction of conditions that need to be satisfied in order to reach these program locations), and the equivalence of the arguments before the calls. This will be demonstrated in two examples later on.

$$\frac{\begin{array}{ll}(7.1) & \forall\langle F,G\rangle \in \mathit{map}.\ \{\\ (7.2) & \quad \mathit{comp\text{-}equiv}(F,G) \wedge \\ (7.3) & \quad \vdash_{\mathbb{L}_{\mathbb{UP}}} \ \mathit{reach\text{-}equiv}(F^{UP},G^{UP})\ \}\end{array}}{(7.4) \quad \forall\langle F,G\rangle \in \mathit{map}.\ \mathit{mutual\text{-}terminate}(F,G)} \ (\text{M-TERM}) \qquad (7)$$

**Fig. 9** Rule (M- TERM): An inference rule for proving the mutual termination of procedures. Note that Premise 7.2 can be proven by the (PROC- P- EQ) rule

## 4.2 Rule (M- TERM)

The mutual termination rule (M- TERM) is stated in Fig. 9. It is interesting to note that unlike proofs of procedure termination, here we do not rely on well-founded sets (see, for example, [6], Sect. 3.4).

*Example 7* Continuing Example 5, we now prove the mutual termination of the two programs in Fig. 4. Since we already proved $\mathit{comp\text{-}equiv}(gcd_1, gcd_2)$ in Example 5, it is left to check Premise (3.3), i.e.,

$$\vdash_{\mathbb{L}_{\mathbb{UP}}} \ \mathit{reach\text{-}equiv}(gcd_1{}^{UP}, gcd_2{}^{UP}).$$

Since in this case we only have a single procedure call in each side, the only thing we need to check in order to establish reach-equivalence, is that the guards controlling their calls are equivalent, and that they are called with the same input arguments. The verification condition is thus:

$$\begin{array}{ll}(T_{gcd_1} \wedge T_{gcd_2} \wedge (a = x) \wedge (b = y)) \rightarrow & \\ (\ ((y_0 > 0) \leftrightarrow (b_0 \neq 0)) \wedge & \mathit{//Equal\ guards} \\ \ ((y_0 > 0) \rightarrow ((b_0 = y_0) \wedge (a_1 = z_0 \mod y_0)))\ ) & \mathit{//Equal\ inputs}\end{array} \qquad (8)$$

where $T_{gcd_1}$ and $T_{gcd_2}$ are as defined in Eq. (4) and (5). □

## 4.3 Rule (M- TERM) is sound

We now prove the following:

**Theorem 2 (Soundness)** *If the proof system* $\mathbb{L}_{\mathbb{UP}}$ *is sound then the rule* (M- TERM) *is sound.*

*Proof* In case the computations of $P_1$ and $P_2$ are both finite or both infinite the consequent of the rule holds by definition. It is left to consider the case in which one of the computations is finite and the other is infinite. We show that if the premise of (M- TERM) holds such a case is impossible.

Let $P_1$ and $P_2$ be two programs in LPL, $\pi_1$ and $\pi_2$ maximal subcomputations from some levels in $P_1$ and $P_2$, respectively, $t_1$ and $t_2$ the stack-level trees of these computations, $n_1$ and $n_2$ the root nodes of $t_1$ and $t_2$ respectively and $F = p[n_1]$ and $G = p[n_2]$. Assume $\pi_1$ and $\pi_2$ are argument-equivalent. Without loss of generality assume also that $\pi_1$ is finite and $\pi_2$ is not.

Consider the computation $\pi_1$. We continue as in the proof of Theorem 1. We construct a computation $\pi_1'$ in $F^{UP}$, which is the same as $\pi_1$ in the level of $n_1$, with the following change. Each closed subcomputation at a deeper level caused by a call $c_F$, is replaced with a subcomputation through an uninterpreted procedure UP($\mathit{callee}[c_F]$), which receives and returns the same values as received and returned by $c_F$. In a similar way we construct a computation $\pi_2'$

in $G^{UP}$ corresponding to $\pi_2$. By Premise (7.2), any pair of calls to some procedures $p_1$ and $p_2$ (related by *map*) satisfy $comp\text{-}equiv(p_1, p_2)$. Thus, any pair of calls to an uninterpreted procedure UP($p_1$) (which is equal to UP($p_2$)) in $\pi'_1$ and $\pi'_2$ satisfy the congruence condition (see (1)). As in the proof of the (PROC- P- EQ) rule, this is sufficient to conclude that $\pi'_1$ and $\pi'_2$ are valid subcomputations through $F^{UP}$ and $G^{UP}$ (but not necessarily closed).

By Premise (7.3) of the rule and the soundness of the underlying proof system $\mathbb{L}_{\mathbb{UP}}$, $\pi'_1$ and $\pi'_2$ satisfy the condition in $reach\text{-}equiv(F^{UP}, G^{UP})$ . It is left to show that this implies that $\pi_2$ must be finite. We will prove this fact by induction on the depth $d$ of $t_1$.

*Base* $d = 0$. In this case $n_1$ is a leaf and $\pi_1$ does not execute any **call** statements. Assume that $\pi_2$ executes some call statement $c_G$ in $G$. Since by Premise (7.3) $reach\text{-}equiv(F^{UP}, G^{UP})$ holds, then there must be some call $c_F$ in $F$ such that $\langle callee[c_F], callee[c_G]\rangle \in map$ and some configuration $C_1 \in \pi'_1$ such that current-label$[C_1]$ = before$[c_F]$ (i.e., $\pi'_1$ reaches the $c_F$ call). But this is impossible as $n_1$ is a leaf. Thus $\pi_2$ cannot be infinite.

*Step*

1. Assume (by the induction hypothesis) that if $\pi_1$ is a finite computation with stack-level tree $t_1$ of depth $d < i$ then any $\pi_2$ such that

$$\text{first}[\pi_1].\sigma[\,\overline{arg\text{-}r_F}\,] = \text{first}[\pi_2].\sigma[\,\overline{arg\text{-}r_G}\,],$$

   cannot be infinite. We now prove this for $\pi_1$ with $t_1$ of depth $d = i$.

2. Let $\widehat{\pi}_2$ be some subcomputation of $\pi_2$ from level $d[n_2] + 1$, $C_2$ be the configuration in $\pi_2$ which comes immediately before $\widehat{\pi}_2$ ($C_2$ = pred$[\widehat{\pi}_2]$). Let $c_G$ be the **call** statement in $G$ which is executed at $C_2$ (in other words current-label$[C_2]$ = before$[c_G]$).
3. Since by Premise (7.3) $reach\text{-}equiv(F^{UP}, G^{UP})$, there must be some call $c_F$ in $F$ such that $\langle callee[c_F], callee[c_G]\rangle \in map$ and some configuration $C_1 \in \pi'_1$ from which the call $c_F$ is executed (i.e., current-label$[C_1]$ = before$[c_F]$), and $C_1$ passes the same input argument values to $c_F$ as $C_2$ to $c_G$. In other words, if $c_F$ = **call** $p_1(\overline{e}_1; \overline{x}_1)$ and $c_G$ = **call** $p_2(\overline{e}_2; \overline{x}_2)$, then $C_1.\sigma[\,\overline{e_1}\,] = C_2.\sigma[\,\overline{e_2}\,]$. But then, there is a subcomputation $\widehat{\pi}_1$ of $\pi_1$ from level $d[n_1] + 1$ which starts immediately after $C_1$ ($C_1$ = pred$[\widehat{\pi}_1]$]).
4. $\widehat{\pi}_1$ is finite because $\pi_1$ is finite. The stack-level tree $\widehat{t_1}$ of $\widehat{\pi}_1$ is a subtree of $t_1$ and its depth is less than $i$. Therefore, by the induction hypothesis (the assumption in item 1) $\widehat{\pi}_2$ must be finite as well.
5. In this way, all subcomputations of $\pi_2$ *from* level $d[n_2] + 1$ are finite. By definition, all subcomputations of $\pi_2$ *at* level $d[n_2]$ are finite. Therefore $\pi_2$ is finite. □

## 4.4 Using rule (M- TERM): a long example

In this example we set the domain $\mathbb{D}$ to be the set of binary trees with natural values in the leafs and the + and * operators at internal nodes.[6]

Let $t_1, t_2 \in \mathbb{D}$. We define the following operators:

- isleaf($t_1$) returns TRUE if $t_1$ is a leaf and FALSE otherwise.
- isplus($t_1$) returns TRUE if $t_1$ has "+" in its root node and FALSE otherwise.
- leftson($t_1$) returns FALSE if $t_1$ is a leaf, and the tree which is its left son otherwise.

[6] To be consistent with the definition of LPL (Definition 1), the domain must also include TRUE and FALSE. Hence we also set the constants TRUE and FALSE to be the leafs with 1 and 0 values, respectively.

```
procedure $Eval_1$(val a; ret r):
    if isleaf($a$) then
        r := a
    else
        if isplus($a$) then
            $l_1$ call $Plus_1$(a; r) $l_3$
        else
            if ismult($a$) then
                $l_5$ call $Mult_1$(a; r) $l_7$
            fi
        fi
    fi
    return

procedure $Plus_1$(val a; ret r):
    $l_9$ call $Eval_1$(leftson($a$); $v$);
    $l_{11}$ call $Eval_1$(rightson($a$); $u$);
    r := doplus($v, u$);
    return

procedure $Mult_1$(val a; ret r):
    $l_{13}$ call $Eval_1$(leftson($a$); $v$);
    $l_{15}$ call $Eval_1$(rightson($a$); $u$);
    r := domult($v, u$);
    return
```

```
procedure $Eval_2$(val x; ret y):
    if isleaf($x$) then
        y := x
    else
        if ismult($x$) then
            $l_2$ call $Mult_2$(x; y) $l_4$
        else
            if isplus($x$) then
                $l_6$ call $Plus_2$(x; y) $l_8$
            fi
        fi
    fi
    return

procedure $Plus_2$(val x; ret y):
    $l_{10}$ call $Eval_2$(rightson($x$); $w$);
    $l_{12}$ call $Eval_2$(leftson($x$); $z$);
    y := doplus($w, z$);
    return

procedure $Mult_2$(val x; ret y):
    $l_{14}$ call $Eval_2$(rightson($x$); $w$);
    $l_{16}$ call $Eval_2$(leftson($x$); $z$);
    y := domult($w, z$);
    return
```

**Fig. 10** Two procedures to calculate the value of an expression tree. Only labels around the **call** constructs are shown

– doplus($l_1, l_2$) returns a leaf with a value equal to the sum of the values in $l_1$ and $l_2$, if $l_1$ and $l_2$ are leafs, and FALSE otherwise.

The operators ismult($t_1$), rightson($t_1$) and domult($t_1, t_2$) are defined similarly to isplus, leftson and doplus, respectively.

The two procedures in Fig. 10 calculate the value of an expression tree.

We introduce three uninterpreted procedures $E$, $P$ and $M$ and set the mapping UP to satisfy

$$\begin{aligned}\mathrm{UP}(Eval_1) &= \mathrm{UP}(Eval_2) = E,\\ \mathrm{UP}(Plus_1) &= \mathrm{UP}(Plus_2) = P,\\ \mathrm{UP}(Mult_1) &= \mathrm{UP}(Mult_2) = M.\end{aligned}$$

The SSA form of the formulas which represent the possible computations of isolated procedure bodies are:

$$T_{Eval_1} = \begin{pmatrix} a_0 = a & \wedge \\ (isleaf(a_0) \to r_1 = a_0) & \wedge \\ (\neg\, isleaf(a_0) \wedge isplus(a_0) \to P(a_0, r_1)) & \wedge \\ (\neg\, isleaf(a_0) \wedge \neg\, isplus(a_0) \wedge ismult(a_0) \to M(a_0, r_1)) & \wedge \\ r = r_1 & \end{pmatrix}$$

$$T_{Eval_2} = \begin{pmatrix} x_0 = x & \wedge \\ (isleaf(x_0) \to y_1 = x_0) & \wedge \\ (\neg\, isleaf(x_0) \wedge ismult(x_0) \to M(x_0, y_1)) & \wedge \\ (\neg\, isleaf(x_0) \wedge \neg\, ismult(x_0) \wedge isplus(x_0) \to P(x_0, y_1)) & \wedge \\ y = y_1 & \end{pmatrix}$$

$$T_{Plus_1} = \begin{pmatrix} a_0 = a & \wedge \\ E(leftson(a_0), v_1) & \wedge \\ E(rightson(a_0), u_1) & \wedge \\ r_1 = doplus(v_1, u_1) & \wedge \\ r = r_1 & \end{pmatrix} \quad T_{Plus_2} = \begin{pmatrix} x_0 = x & \wedge \\ E(rightson(x_0), w_1) & \wedge \\ E(leftson(x_0), z_1) & \wedge \\ y_1 = doplus(w_1, z_1) & \wedge \\ y = y_1 & \end{pmatrix}$$

$$T_{Mult_1} = \begin{pmatrix} a_0 = a & \wedge \\ E(leftson(a_0), v_1) & \wedge \\ E(rightson(a_0), u_1) & \wedge \\ r_1 = domult(v_1, u_1) & \wedge \\ r = r_1 & \end{pmatrix} \quad T_{Mult_2} = \begin{pmatrix} x_0 = x & \wedge \\ E(rightson(x_0), w_1) & \wedge \\ E(leftson(x_0), z_1) & \wedge \\ y_1 = domult(w_1, z_1) & \wedge \\ y = y_1 & \end{pmatrix}$$

Proving partial computational equivalence for each of the procedure pairs amounts to proving the following formulas to be valid:

$$(a = x \wedge T_{Eval_1} \wedge T_{Eval_2}) \rightarrow r = y$$
$$(a = x \wedge T_{Plus_1} \wedge T_{Plus_2}) \rightarrow r = y$$
$$(a = x \wedge T_{Mult_1} \wedge T_{Mult_2}) \rightarrow r = y.$$

To prove these formulas it is enough for $L_{UF}$ to know the following facts about the operators of the domain:

$$\forall l_1, l_2 (doplus(l_1, l_2) = doplus(l_2, l_1) \wedge domult(l_1, l_2) = domult(l_2, l_1))$$
$$\forall t_1 (isleaf(t_1) \rightarrow \neg\, isplus(t_1) \wedge \neg\, ismult(t_1))$$
$$\forall t_1 (isplus(t_1) \rightarrow \neg\, ismult(t_1) \wedge \neg\, isleaf(t_1))$$
$$\forall t_1 (ismult(t_1) \rightarrow \neg\, isleaf(t_1) \wedge \neg\, isplus(t_1))$$

This concludes the proof of partial computational equivalence using rule (PROC- P- EQ). To prove mutual termination using the (M- TERM) rule we need in addition to verify reach-equivalence of each pair of procedures.

To check reach-equivalence we should check that the guards and the read arguments at labels of related calls are equivalent. This can be expressed by the following formulas:

$$\begin{aligned} \varphi_1 = (\ & g_1 = (\neg\, isleaf(a_0) \wedge isplus(a_0)) & \wedge \\ & g_2 = (\neg\, isleaf(x_0) \wedge \neg\, ismult(x_0) \wedge isplus(x_0)) & \wedge \\ & g_3 = (\neg\, isleaf(a_0) \wedge \neg\, isplus(a_0) \wedge ismult(a_0)) & \wedge \\ & g_4 = (\neg\, isleaf(x_0) \wedge ismult(x_0)) & \wedge \\ & g_1 \leftrightarrow g_2 & \wedge \\ & g_3 \leftrightarrow g_4 & \wedge \\ & g_1 \rightarrow a_0 = x_0 & \wedge \\ & g_3 \rightarrow a_0 = x_0) & \end{aligned}$$

The guards at all labels in $Plus_1$, $Plus_2$, $Mult_1$ and $Mult_2$ are all true, therefore the reach-equivalence formulas for these procedures collapse to:

$$\begin{aligned} \varphi_2 = \varphi_3 = (\ & (leftson(a_0) = rightson(x_0) \vee leftson(a_0) = leftson(x_0)) & \wedge \\ & (rightson(a_0) = rightson(x_0) \vee rightson(a_0) = leftson(x_0)) & \wedge \\ & (leftson(a_0) = rightson(x_0) \vee rightson(a_0) = rightson(x_0)) & \wedge \\ & (leftson(a_0) = leftson(x_0) \vee rightson(a_0) = leftson(x_0))) & \end{aligned}$$

In this formula each call in each side is mapped to one of the calls on the other side: the first two lines map calls of side one to calls on side two, and the last two lines map calls of side two to side one. Finally, the formulas that need to be validated are:

$$(a = x \wedge T_{Eval_1} \wedge T_{Eval_2}) \rightarrow \varphi_1$$
$$(a = x \wedge T_{Plus_1} \wedge T_{Plus_2}) \rightarrow \varphi_2$$
$$(a = x \wedge T_{Mult_1} \wedge T_{Mult_2}) \rightarrow \varphi_3.$$

## 5 A proof rule for equivalence of reactive programs

Rules (PROC- P- EQ) and (M- TERM) that we studied in the previous two sections, are concerned with equivalence of finite programs, and with proving the mutual termination of programs, respectively. In this section we introduce a rule that generalizes (PROC- P- EQ) in the sense that it is not restricted to finite computations. This generalization is necessary for *reactive* programs. We say that two reactive procedures $F$ and $G$ are *reactively equivalent* if given the same input sequences, their output sequences are the same.

### 5.1 Definitions

The introduction of the rule and later on the proof requires an extension of LPL to allow input and output constructs:

**Definition 15 (LPL with I/O constructs (LPL+IO))** LPL+IO is the LPL programming language with two additional statement constructs:

$$\textbf{input}(x) \mid \textbf{output}(e)$$

where $x \in V$ is a variable and $e$ is an expression over $O_{\mathbb{D}}$. If a sequence of **input** constructs appear in a procedure they must appear before any other statement in that procedure. This fact is important for the proof of correctness.[7] □

The input and output constructs are assumed to read and write values in the domain $\mathbb{D}$. A reactive system reads a sequence of inputs using its **input** constructs and writes a sequence of outputs by its **output** constructs. These sequences may be finite or infinite. A computation of an LPL+IO program is a sequence of configurations of the form: $C = \langle d, O, \overline{pc}, \sigma, \overline{R}, \overline{W}\rangle$ where $\overline{R}$ and $\overline{W}$ are sequences of values in $\mathbb{D}$ and all other components in $C$ are as in Sect. 2.2. Intuitively, $\overline{R}$ denotes the suffix of the sequence of inputs that remains to be read after configuration $C$, and $\overline{W}$ is the sequence of outputs that were written until configuration $C$.

**Definition 16 (Transition relation in LPL+IO)** Let "$\rightarrow$" be the least relation among configurations which satisfies: if $C \rightarrow C'$, $C = \langle d, O, \overline{pc}, \sigma, \overline{R}, \overline{W}\rangle$, $C' = \langle d', O', \overline{pc}', \sigma', \overline{R}', \overline{W}'\rangle$ then:

1. If current-label$[C]$ = before$[S]$ for some input construct $S$ = "**input**$(x)$", and $R_0$ is the value being read, then $d' = d$, $O' = O$, $\overline{pc}' = \langle pc_i\rangle_{i\in\{0,\ldots,d-1\}} \cdot \langle$after$[S]\rangle$, $\sigma' = \sigma[R_0|x]$, $\overline{R}' = \langle R_i\rangle_{i\in\{1,\ldots\}}$, $\overline{W}' = \overline{W}$.

[7] A procedure that reads inputs during its execution rather than at its beginning can be simulated by replacing the input command with a procedure call. The called procedure only reads the inputs and returns them to the caller, and hence respects the requirement that the inputs are read at its beginning.

2. If current-label$[C]$ = before$[S]$ for some output construct $S =$ "**output**$(e)$" then $d' = d,\ O' = O,\ \overline{pc}' = \langle pc_i \rangle_{i \in \{0,\dots,d-1\}} \cdot \langle \text{after}[S] \rangle,\ \sigma' = \sigma,\ \overline{R}' = \overline{R},\ \overline{W}' = \overline{W} \cdot \langle \sigma[e] \rangle$

and for all other statement constructs the transition relation is defined as in Definition 4. □

By definition of the transition relation of LPL+IO, the $\overline{W}$ sequence of a configuration contains the $\overline{W}$ sequence of each of its predecessors as a prefix. We say that the *input sequence of a computation* is the $\overline{R}$ sequence of its first configuration. If the computation is finite then *its output sequence* is the $\overline{W}$ sequence of its last configuration. If the computation is infinite then its *output sequence* is the supremum of the $\overline{W}$ sequences of all its configurations, when we take the natural containment order between sequences (i.e., the sequence that contains each $\overline{W}$ sequence as its prefix). For a computation $\pi$, we denote by InSeq$[\pi]$ its input sequence and by OutSeq$[\pi]$ its output sequence. For a finite computation $\pi$, we denote by $\Delta\overline{R}[\pi]$ the inputs consumed along $\pi$, and by $\Delta\overline{W}[\pi]$ the outputs written during $\pi$.

**Definition 17 (input-equivalence of subcomputations with respect to procedures)** Two subcomputations $\pi_1'$ and $\pi_2'$ that are argument-equivalent with respect to two procedures $F$ and $G$ are called *input equivalent* if

$$\text{first}[\pi_1'].\overline{R} = \text{first}[\pi_2'].\overline{R}.$$

□

In other words, two subcomputations are input equivalent with respect to procedures $F$ and $G$ if they start at the beginnings of $F$ and $G$ respectively with equivalent read arguments and equivalent input sequences.

We will use the following notations in this section, for procedures $F$ and $G$. The formal definitions of these terms appear in Appendix A.

1. *reactive-equiv*$(F, G)$—Every two input-equivalent subcomputations with respect to $F$ and $G$ generate equivalent output sequences until returning from $F$ and $G$, or forever if they do not return.
2. *return-values-equiv*$(F, G)$—The last configurations of every two input-equivalent *finite* subcomputations with respect to $F$ and $G$ that end in the return from $F$ and $G$, valuate equally the write-arguments of $F$ and $G$, respectively. (note that *return-values-equiv*$(F, G)$ is the same as *comp-equiv*$(F, G)$ other than the fact that it requires the subcomputations to be input equivalent and not only argument equivalent).
3. *input-suffix-equiv*$(F, G)$—Every two input-equivalent *finite* subcomputations with respect to $F$ and $G$ that end at the return from $F$ and $G$, have (at return) the same remaining sequence of inputs.
4. *call-output-seq-equiv*$(F, G)$—Every two input-equivalent subcomputations with respect to $F$ and $G$, generate the same sequence of procedure calls and **output** statements, where corresponding procedure calls are called with equal inputs (read-arguments and input sequences), and **output** statements output equal values.

## 5.2 Rule (REACT- EQ)

Figure 11 presents rule (REACT- EQ), which can be used for proving equivalence of reactive procedures.

$$\frac{\begin{array}{lll}(9.1) & \forall\langle F,G\rangle\in map.\ \{ & \\ (9.2) & \vdash_{\mathbb{L}_{\mathbb{UP}}} & return\text{-}values\text{-}equiv(F^{UP},G^{UP})\ \wedge \\ (9.3) & \vdash_{\mathbb{L}_{\mathbb{UP}}} & input\text{-}suffix\text{-}equiv(F^{UP},G^{UP})\ \wedge \\ (9.4) & \vdash_{\mathbb{L}_{\mathbb{UP}}} & call\text{-}output\text{-}seq\text{-}equiv(F^{UP},G^{UP})\ \}\end{array}}{(9.5)\quad \forall\langle F,G\rangle\in map.\ reactive\text{-}equiv(F,\ G)}\ (\text{REACT-EQ}) \qquad (9)$$

**Fig. 11** Rule (REACT- EQ): An inference rule for proving the reactive equivalence of procedures

### 5.3 Rule (REACT- EQ) is sound

**Theorem 3 (Soundness)** *If the proof system* $\mathbb{L}_{\mathbb{UP}}$ *is sound then rule* (REACT- EQ) *is sound.*

*Proof* In the following discussion we use the following notation:

| | |
|---|---|
| $P_1, P_2$ | programs in LPL+IO |
| $\pi, \tau$ | subcomputations in $P_1$ and $P_2$, respectively, |
| $t_1, t_2$ | stack-level trees of $\pi$ and $\tau$, respectively, |
| $n_1, n_2$ | the root nodes of $t_1$ and $t_2$, respectively, |
| $F = p[n_1], G = p[n_2]$ | the procedures associated with $n_1$ and $n_2$, |
| $d_1 = d[n_1], d_2 = d[n_2]$ | the levels of nodes $n_1$ and $n_2$, respectively. |

We assume that $\pi$ and $\tau$ are input equivalent and that $\langle F, G\rangle \in map$.

Our main lemma below focuses on finite stack-level trees. The extension to infinite computations will be discussed in Lemma 5.

**Lemma 1**

*If*

(1) $\pi$ *and* $\tau$ *are* maximal *subcomputations,*
(2) $\pi$ *and* $\tau$ *are input equivalent,*
(3) *first*$[\pi].\overline{W} =$ *first*$[\tau].\overline{W}$,
(4) $\pi$ *is finite and its stack-level tree depth is d, and*
(5) *the premises of* (REACT- EQ) *hold,*

*then*

(1) $\tau$ *is finite and its stack-level tree depth is at most d,*
(2) *last*$[\pi].\sigma[\,\overline{arg\text{-}w_F}\,] =$ *last*$[\tau].\sigma[\,\overline{arg\text{-}w_G}\,]$,
(3) *last*$[\pi].\overline{R} =$ *last*$[\tau].\overline{R}$, *and*
(4) *last*$[\pi].\overline{W} =$ *last*$[\tau].\overline{W}$.

While the lemma refers to $\pi$ in the premise and $\tau$ in the consequent, this is done without loss of generality. If premise 1 or 2 is false, the rule holds trivially. Premise 3 holds trivially for the main procedures, and premise 4 holds trivially for the finite computations case (which this lemma covers) for some $d$. Note that consequent 4 implies the consequent of rule (REACT-EQ). Hence proving this lemma proves also Theorem 3 for the case of finite stack-level trees. Together with Lemma 5 that refers to infinite computations, this will prove Theorem 3.

*Proof* (Lemma 1) By induction on the stack-level tree depth $d$.

*Base* $n_1$ is a leaf. Since Premise (9.4) holds, $\tau$ does not contain any calls from $G^{UP}$. Thus, $n_2$ is a leaf as well, and the depth of $t_2$ must be 1 (consequent 1). $\pi$ and $\tau$ contain no calls to procedures, which implies that they are also valid computations through $F^{UP}$ and $G^{UP}$, respectively. Consequents 2, 3 and 4 of the lemma are implied directly from the three premises of (REACT- EQ), respectively, and the soundness of the proof system $\mathbb{L}_{\mathbb{UP}}$.

*Step* We now assume that Lemma 1 holds for all callees of $F$ and $G$ (the procedures of the children in the stack-level trees) and prove it for $F$ and $G$.

The computation $\pi$ is an interleaving between "at-level" and "from-level" subcomputations. We denote the former subcomputations by $\bar{\pi}_i$ for $i \geq 1$, and the latter by $\hat{\pi}_j$ for $j \geq 1$. For example, the subcomputation corresponding to level $d+1$ in the left drawing of Fig. 7 has three "at-level" segments that we number $\bar{\pi}_1, \bar{\pi}_2, \bar{\pi}_3$ (corresponding to segments 2, 4, 8 in the drawing) and two "from-level" subcomputations that we number $\hat{\pi}_1, \hat{\pi}_2$ (segments 3 and 5–7 in the drawing).

Derive a computation $\pi'$ in $F^{UP}$ from $\pi$ as follows. First, set first$[\pi']$ = first$[\pi]$. Further, the at-level subcomputations remain the same (other than the $\overline{R}$ and $\overline{W}$ values—see below) and are denoted respectively $\bar{\pi}'_i$. In contrast the "from-level" subcomputations are replaced as follows. Replace in $\pi$ each subcomputation $\hat{\pi}_j \in from(\pi, d_1)$ caused by a call $c_F$, with a subcomputation $\hat{\pi}'_j$ through an uninterpreted procedure UP$(callee[c_F])$, which returns the same value as returned by $c_F$. A small adjustment to $\overline{R}$ and $\overline{W}$ in $\pi'$ is required since the uninterpreted procedures do not consume inputs or generate outputs. Hence, $\overline{R}$ remains constant in $\pi'$ after passing the **input** statements in level $d_1$ and $\overline{W}$ contains only the output values emitted by the at-level subcomputations. In a similar way construct a computation $\tau'$ in $G^{UP}$ corresponding to $\tau$.

In the course of the proof we will show that $\pi'$ and $\tau'$ are valid subcomputations through $F^{UP}$, $G^{UP}$, as they satisfy the congruence condition.

*Proof plan* Proving the step requires several stages. First, we will prove two additional lemmas: Lemma 2 will prove certain properties of "at-level" subcomputations, whereas Lemma 3 will establish several properties of "from-level" subcomputations, assuming the induction hypothesis of Lemma 1. Second, using these lemmas we will establish in Lemma 4 the relation between the beginning and end of subcomputations $\pi$ and $\tau$. This will prove the step of Lemma 1.

The notations in the following lemma correspond to the left drawing in Fig. 12. Specifically,

$$A_1 = \text{first}[\bar{\pi}_i],\ A'_1 = \text{first}[\bar{\pi}'_i],\ A'_2 = \text{first}[\bar{\tau}'_i],\ A_2 = \text{first}[\bar{\tau}_i],$$

$$B_1 = \text{last}[\bar{\pi}_i],\ B'_1 = \text{last}[\bar{\pi}'_i],\ B'_2 = \text{last}[\bar{\tau}'_i],\ B_2 = \text{last}[\bar{\tau}_i].$$

The figure shows the at-level segments $\bar{\pi}_i, \bar{\pi}'_i, \bar{\tau}_i, \bar{\tau}'_i$, the equivalences between various values in their initial configurations which we assume as premises in the lemma, and the equivalences that we prove to hold in the last configurations of these subcomputations.

The at-level segment $\bar{\pi}_i$ may end with some statement **call** $p_1(\overline{e}_1; \overline{x}_1)$ or at a return from procedure $F$. Similarly, $\bar{\tau}_i$ may end with some statement **call** $p_2(\overline{e}_2; \overline{x}_2)$ or at a return from procedure $G$.

**Lemma 2 (Properties of an at-level subcomputation)** *For each i, with respect to* $\bar{\pi}_i$, $\bar{\tau}_i$, $\bar{\pi}'_i$ *and* $\bar{\tau}'_i$,

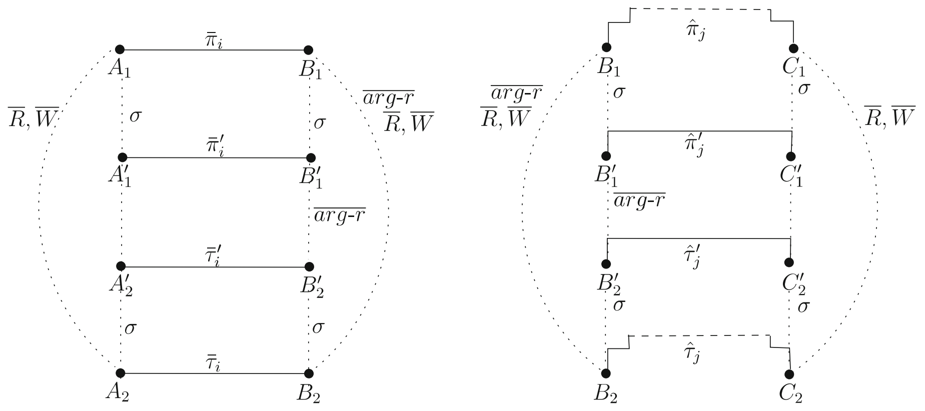

**Fig. 12** (*left*) "at-level" subcomputations—a diagram for Lemma 2. (*right*) "from-level" subcomputations—a diagram for Lemma 3

*if*

1) $A_1.\sigma|_{d_1} = A'_1.\sigma|_{d_1}$
2) $A_2.\sigma|_{d_2} = A'_2.\sigma|_{d_2}$
3) $A_1.\overline{R} = A_2.\overline{R}$
4) $A_1.\overline{W} = A_2.\overline{W}$
5) *If* $i = 1$ *then* $(A_1.\overline{R} = A'_1.\overline{R}$ *and* $A_2.\overline{R} = A'_2.\overline{R})$
6) *If* $i = 1$ *then* $A_1.\sigma|_{d_1} = A_2.\sigma|_{d_2}$.

*then*

1) $B_1.\sigma|_{d_1} = B'_1.\sigma|_{d_1}$
2) $B_2.\sigma|_{d_2} = B'_2.\sigma|_{d_2}$
3) *If* $\bar{\pi}_i$ *ends with* **call** $p_1(\bar{e}_1; \bar{x}_1)$ *then* $\bar{\tau}_i$ *ends with* **call** $p_2(\bar{e}_2; \bar{x}_2)$ *and* $\langle p_1, p_2\rangle \in map$
4) *If* $\bar{\pi}'_i$ *ends with a* **call** *statement, then* $B'_1.\sigma[\bar{e}_1] = B'_2.\sigma[\bar{e}_2]$
5) *If* $\bar{\pi}_i$ *ends with a* **call** *statement, then* $B_1.\sigma[\bar{e}_1] = B_2.\sigma[\bar{e}_2]$
6) $B_1.\overline{R} = B_2.\overline{R}$
7) $B_1.\overline{W} = B_2.\overline{W}$

(*In Fig.* 12 *consequents* 4, 5 *are represented by the requirement of having equal* $\overline{arg\text{-}r}$ *values* (*equal formal parameters*)).

*Proof* (Lemma 2)

1. (Consequent 1) For $i > 1$: $A_1.\sigma|_{d_1} = A'_1.\sigma|_{d_1}$(Premise 1), hence, by definition of $\bar{\pi}'_i$ (which implies that $\bar{\pi}_i$ and $\bar{\pi}'_i$ are equivalent, because they are defined by the same LPL code and begin with the same variable values), we have $B_1.\sigma|_{d_1} = B'_1.\sigma|_{d_1}$.
   Recall that by definition of LPL+IO, **input** statements may appear only at the beginning of the procedure. Therefore, for $i = 1$ it is a little more complicated because of possible **input** statements. In addition to Premise 1 we now also need $A_1.\overline{R} = A'_1.\overline{R}$ (Premise 5) and again, by definition of $\bar{\pi}'_i$ we have $B_1.\sigma|_{d_1} = B'_1.\sigma|_{d_1}$.
2. (Consequent 2) Dual to the proof of consequent 1, using Premise 2 instead of Premise 1.

3. Since $\bar{\pi}_i, \bar{\pi}'_i$ are defined by the same LPL code and begin with the same variable values and, for the case of $i = 1$, the same input sequence, they consume the same portions of the input sequence and produce the same output subsequence. Thus, we have $\Delta\overline{W}[\bar{\pi}_i] = \Delta\overline{W}[\bar{\pi}'_i]$, $\Delta\overline{R}[\bar{\pi}_i] = \Delta\overline{R}[\bar{\pi}'_i]$, and in a similar way with respect to $\bar{\tau}_i, \bar{\tau}'_i$, we have $\Delta\overline{W}[\bar{\tau}_i] = \Delta\overline{W}[\bar{\tau}'_i]$, $\Delta\overline{R}[\bar{\tau}_i] = \Delta\overline{R}[\bar{\tau}'_i]$.
4. If $i = 1$, $\bar{\pi}_i, \bar{\pi}'_i, \bar{\tau}'_i$ and $\bar{\tau}_i$ are the first segments in $\pi, \pi', \tau'$ and $\tau$, and thus may contain **input** statements. Then by Premise 5 of the lemma we have $A_1.\overline{R} = A'_1.\overline{R}$, $A_2.\overline{R} = A'_2.\overline{R}$, and by $A_1.\overline{R} = A_2.\overline{R}$ (Premise 3) and transitivity of equality we have $A'_1.\overline{R} = A'_2.\overline{R}$.
5. (Consequents 3 and 4) The subcomputations from the beginning of $\pi'$ to the end of $\bar{\pi}'_i$ and from the beginning of $\tau'$ to the end of $\bar{\tau}'_i$ are prefixes of valid subcomputations through $F^{UP}$ and $G^{UP}$. These subcomputations are input equivalent due to $A'_1.\overline{R} = A'_2.\overline{R}$ (see item 4 above) and $A_1.\sigma|_{d_1} = A_2.\sigma|_{d_2}$ (Premise 6). If $\bar{\pi}_i$ ends with **call** $p_1(\overline{e}_1; \overline{x}_1)$ then $\bar{\pi}'_i$ ends with **call** UP$(p_1)(\overline{e}_1; \overline{x}_1)$. Then, by *call-output-seq-equiv*$(F^{UP}, G^{UP})$ (premise 9.4 of (REACT- EQ)), $\bar{\tau}'_i$ must end with **call** UP$(p_2)(\overline{e}_2; \overline{x}_2)$ where $\langle p_1, p_2\rangle \in map$, and therefore $\bar{\tau}_i$ ends with **call** $p_2(\overline{e}_2; \overline{x}_2)$. This proves consequent 3. The same premise also implies that $B'_1.\sigma[\,\overline{e}_1] = B'_2.\sigma[\,\overline{e}_2]$, which proves consequent 4, and that $\Delta\overline{W}[\bar{\pi}'_i] = \Delta\overline{W}[\bar{\tau}'_i]$.
6. (Consequent 5) Implied by consequents 1, 2 and 4 that we have already proved, and transitivity of equality.
7. Consider $\bar{\pi}'_i$ and $\bar{\tau}'_i$. For $i = 1$, $\bar{\pi}'_1$ and $\bar{\tau}'_1$ are prefixes of valid input-equivalent subcomputations through $F^{UP}$ and $G^{UP}$, and as in any such subcomputation the inputs are consumed only at the beginning. Therefore, *input-suffix-equiv*$(F^{UP}, G^{UP})$ (Premise 9.3), which implies equality of $\Delta\overline{R}$ of these subcomputations, also implies $\Delta\overline{R}[\bar{\pi}'_1] = \Delta\overline{R}[\bar{\tau}'_1]$. For $i > 1$, no input values are read in $\bar{\pi}'_i$ and $\bar{\tau}'_i$ and hence $\Delta\overline{R}[\bar{\pi}'_i] = \Delta\overline{R}[\bar{\tau}'_i] = \emptyset$. Thus, for any $i$ we have $\Delta\overline{R}[\bar{\pi}'_i] = \Delta\overline{R}[\bar{\tau}'_i]$.
8. (Consequent 6) By $\Delta\overline{R}[\bar{\pi}_i] = \Delta\overline{R}[\bar{\pi}'_i]$, $\Delta\overline{R}[\bar{\tau}_i] = \Delta\overline{R}[\bar{\tau}'_i]$ (see item 3), $\Delta\overline{R}[\bar{\pi}'_i] = \Delta\overline{R}[\bar{\tau}'_i]$ (see item 7) and transitivity of equality we have $\Delta\overline{R}[\bar{\pi}_i] = \Delta\overline{R}[\bar{\tau}_i]$. This together with $A_1.\overline{R} = A_2.\overline{R}$ (Premise 3) entails consequent 6.
9. (Consequent 7) By $\Delta\overline{W}[\bar{\pi}_i] = \Delta\overline{W}[\bar{\pi}'_i]$, $\Delta\overline{W}[\bar{\tau}_i] = \Delta\overline{W}[\bar{\tau}'_i]$ (see item 3), $\Delta\overline{W}[\bar{\pi}'_i] = \Delta\overline{W}[\bar{\tau}'_i]$ (see end of item 5) and transitivity of equality we have $\Delta\overline{W}[\bar{\pi}_i] = \Delta\overline{W}[\bar{\tau}_i]$. This together with $A_1.\overline{W} = A_2.\overline{W}$ (Premise 4) entails consequent 7.

(End of proof of Lemma 2). □

The notations in the following lemma corresponds to the right drawing in Fig. 12. The beginning configurations $B_1, B'_1, B'_2, B_2$ are the same as the end configurations of the drawing in the left of the same figure. In addition we now have the configurations at the end of the 'from-level' subcomputations, denoted by $C_1, C'_1, C_2, C'_2$, or, more formally:

$$C_1 = \text{last}[\hat{\pi}_j],\;\; C'_1 = \text{last}[\hat{\pi}'_j],\;\; C'_2 = \text{last}[\hat{\tau}'_j],\;\; C_2 = \text{last}[\hat{\tau}_j].$$

Note that $\hat{\pi}_j$ is finite by definition of $\pi$, and therefore last$[\hat{\pi}_j]$ is well-defined. We will show in the proof of the next lemma that $\hat{\tau}_j$ is finite as well, and therefore last$[\hat{\tau}_j]$ is also well-defined.

**Lemma 3 (Properties of a 'from-level' subcomputation)** *With respect to $\hat{\pi}_j, \hat{\tau}_j, \hat{\pi}'_j$ and $\hat{\tau}'_j$ for some $j$, let current-label$[B_1]$ = before$[c_F]$, current-label$[B_2]$ = before$[c_G]$, $c_F$ =* **call** *$p_1(\overline{e}_1; \overline{x}_1)$, and $c_G$ =* **call** *$p_2(\overline{e}_2; \overline{x}_2)$. Then* *if*

$$\begin{array}{l} 1)\ B_1.\sigma|_{d_1} = B_1'.\sigma|_{d_1} \\ 2)\ B_2.\sigma|_{d_2} = B_2'.\sigma|_{d_2} \\ 3)\ B_1.\sigma[\,\overline{e}_1] = B_2.\sigma[\,\overline{e}_2] \\ 4)\ B_1'.\sigma[\,\overline{e}_1] = B_2'.\sigma[\,\overline{e}_2] \\ 5)\ B_1.\overline{R} = B_2.\overline{R} \\ 6)\ B_1.\overline{W} = B_2.\overline{W} \\ 7)\ \hat{\pi}_j \textit{ has a stack-level tree of depth at most } d-1 \\ 8)\langle p_1, p_2\rangle \in \textit{map}, \end{array}$$

*then*

$$\begin{array}{l} 1)\ C_1.\overline{R} = C_2.\overline{R} \\ 2)\ C_1.\overline{W} = C_2.\overline{W} \\ 3)\ C_1.\sigma|_{d_1} = C_1'.\sigma|_{d_1} \\ 4)\ C_2.\sigma|_{d_2} = C_2'.\sigma|_{d_2} \\ 5)\ \hat{\tau}_j \textit{ has a stack-level tree of depth at most } d-1. \end{array}$$

*Proof* (Lemma 3)

1. (Consequents 1, 2, 5) As $\hat{\pi}_j$ has a stack-level tree of depth at most $d-1$ (Premise 7), by $B_1.\sigma[\,\overline{e}_1] = B_2.\sigma[\,\overline{e}_2]$ (Premise 3), $B_1.\overline{R} = B_2.\overline{R}$ (Premise 5), and the induction hypothesis of Lemma 1, $\hat{\tau}_j$ has stack-level tree of depth at most $d-1$ and: $C_1.\sigma[\,\overline{x}_1\,] = C_2.\sigma[\,\overline{x}_2\,]$, $C_1.\overline{R} = C_2.\overline{R}$ and $\Delta\overline{W}[\hat{\pi}_j] = \Delta\overline{W}[\hat{\tau}_j]$. Therefore, by $B_1.\overline{W} = B_2.\overline{W}$ (Premise 6) we have $C_1.\overline{W} = C_2.\overline{W}$.
2. (Consequents 3, 4) As $\hat{\pi}_j'$ and $\hat{\tau}_j'$ are computations through calls to the same uninterpreted procedure (by premise 8) we can choose them in such a way that they satisfy $C_1'.\sigma[\overline{x}_1] = C_1.\sigma[\overline{x}_1] = C_2.\sigma[\overline{x}_2] = C_2'.\sigma[\overline{x}_2]$, and hence satisfy (1). As valuations of other variables by $\sigma|_{d_1}$ and $\sigma|_{d_2}$ are unchanged by subcomputations in higher levels (above $d_1$ and $d_2$, respectively), we have $C_1.\sigma|_{d_1} = C_1'.\sigma|_{d_1}$ and $C_2.\sigma|_{d_2} = C_2'.\sigma|_{d_1}$.

(End of proof of Lemma 3). □

Using Lemmas 2 and 3 we can establish equivalences of values in the end of input-equivalent subcomputations, based on the fact that every subcomputation, as mentioned earlier, is an interleaving between "at-level" and "from-level" subcomputations.

**Lemma 4 (Properties of subcomputations)** *Let* $A_1 = \textit{first}[\bar{\pi}_i]$, $A_2 = \textit{first}[\bar{\tau}_i]$, $A_1' = \textit{first}[\bar{\pi}_i']$ *and* $A_2' = \textit{first}[\bar{\tau}_i']$ *be the first configurations of* $\bar{\pi}_i, \bar{\tau}_i, \bar{\pi}_i'$ *and* $\bar{\tau}_i'$ *respectively for some* $i$. *Then these configurations satisfy the following conditions:*

$$\begin{array}{l} 1)\ A_1.\sigma|_{d_1} = A_1'.\sigma|_{d_1} \\ 2)\ A_2.\sigma|_{d_2} = A_2'.\sigma|_{d_2} \\ 3)\ A_1.\overline{R} = A_2.\overline{R} \\ 4)\ A_1.\overline{W} = A_2.\overline{W} \end{array}$$

*Proof* By induction on $i$.

*Base* For $i = 1$, $\bar{\pi}_i$ and $\bar{\tau}_i$ start at the beginning of $F$ and $G$. Hence $\bar{\pi}_i'$ and $\bar{\tau}_i'$ are at the beginning of $F^{UP}$ and $G^{UP}$. By the definition of $\bar{\pi}_1'$ and $\bar{\tau}_1'$, the lemma is valid in this case because $\pi$ and $\tau$ are input equivalent (between themselves and with $\pi'$ and $\tau'$). Consequent 4 stems from Premise 3 of Lemma 1.

*Step* Consider in $\pi$ some consecutive at-level and from-level subcomputations $\bar{\pi}_i$ and $\hat{\pi}_j$ and their respective counterparts: $(\bar{\pi}'_i, \hat{\pi}'_j)$ in $\pi'$, $(\bar{\tau}'_i, \hat{\tau}'_j)$ in $\tau'$, and finally $(\bar{\tau}_i, \hat{\tau}_j)$ in $\tau$.

By the induction hypothesis and the finiteness of $\hat{\pi}_i$ (guaranteed by the hypothesis of Lemma 1), premises 1–4 of Lemma 2 hold. Premises 5 and 6 hold as well because they are implied by the definitions of $\pi'$, $\tau'$, $\pi$ and $\tau$. Thus, the premises and therefore the consequents of Lemma 3 hold, which implies that the induction hypothesis of the current lemma holds for $i + 1$.

(End of proof of Lemma 4). □

Consequent 1 of Lemma 1 holds because for any $j$, if the depths of the stack-level trees of $\hat{\tau}_j$ are bounded by $d - 1$ (consequent 5 of Lemma 3) then the depths of the stack-level tree of $\tau$ is bounded by $d$.

The other consequents of Lemma 1 are proved by using Lemma 4. Let $(\bar{\pi}_l,\bar{\tau}_l)$ be the last pair of subcomputations (e.g., in the left drawing of Fig. 7, segment 8 is the last in level $d+1$). Their counterparts in the isolated bodies $F^{UP}$ and $G^{UP}$, $\bar{\pi}'_l$ and $\bar{\tau}'_l$, are the last parts of the computations $\pi'$ and $\tau'$. We use the same notation as before for denoting the configurations in the end of these subcomputations:

$$B_1 = \text{last}[\pi],\ B'_1 = \text{last}[\pi'],\ B'_2 = \text{last}[\tau'],\ B_2 = \text{last}[\tau].$$

Therefore $\mathit{return\text{-}values\text{-}equiv}(F^{UP}, G^{UP})$ entails

$$B'_1.\sigma[\,\overline{arg\text{-}w_F}\,] = B'_2.\sigma[\,\overline{arg\text{-}w_G}\,].$$

By Lemma 4 the configurations $A_1 = \text{first}[\bar{\pi}_l]$, $A'_1 = \text{first}[\bar{\pi}'_l]$, $A'_2 = \text{first}[\bar{\tau}'_l]$, and $A_2 = \text{first}[\bar{\tau}_l]$ satisfy the Premises of Lemma 2. By consequents 1 and 2 of this lemma we have $B_1.\sigma|_{d_1} = B'_1.\sigma|_{d_1}$ and $B_2.\sigma|_{d_2} = B'_2.\sigma|_{d_2}$, and by transitivity $B_1.\sigma[\,\overline{arg\text{-}w_F}\,] = B_2.\sigma[\,\overline{arg\text{-}w_G}\,]$ (this proves consequent 2 of Lemma 1). Further, consequents 5 and 6 of Lemma 2 yield $B_1.\overline{R} = B_2.\overline{R}$ and $B_1.\overline{W} = B_2.\overline{W}$ (this proves consequents 3, 4 of Lemma 1).

(End of proof of Lemma 1). □

It is left to consider the case when $\pi$ and $\tau$ are infinite (recall that by Lemma 1 $\pi$ is infinite if and only if $\tau$ is infinite). Hence, there is exactly one infinite branch in each of the stack-level trees $t_1$ and $t_2$ (the stack-level trees of $\pi$ and $\tau$ respectively). Figure 13 presents a possible part of $\pi$ and its corresponding stack-level tree. Consider these infinite branches as infinite sequences of nodes, which begin at their respective roots and continue to nodes of higher levels.

We first need the following definition:

**Definition 18 (Call configuration)** A configuration $C$ is a *call configuration* if current-label $[C] = \text{before}[\textbf{call}\ p(\overline{e}; \overline{x})]$ for some procedure $p$. □

**Lemma 5** *Let $\pi$ and $\tau$ be input-equivalent infinite computations through F and G, respectively, with corresponding stack-level trees $t_1$ and $t_2$. Consider the series of call configurations which are the last in their respective levels on the infinite branches of $t_1$ and $t_2$ (i.e., the ones that bring the execution from one node of the infinite branch to the next one). Let $B_1$ and $B_2$ be a pair of such configurations such that $B_1.d = B_2.d$ and let current-label$[B_1] = \mathit{before}[\textbf{call}\ p_1(\overline{e}_1; \overline{x}_1)]$ and current-label$[B_2] = \mathit{before}[\textbf{call}\ p_2(\overline{e}_2; \overline{x}_2)]$. Then*

1) $\langle p_1, p_2\rangle \in \mathit{map}$
2) $B_1.\sigma[\,\overline{e}_1] = B_2.\sigma[\,\overline{e}_2]$
3) $B_1.\overline{R} = B_2.\overline{R}$
4) $B_1.\overline{W} = B_2.\overline{W}$.

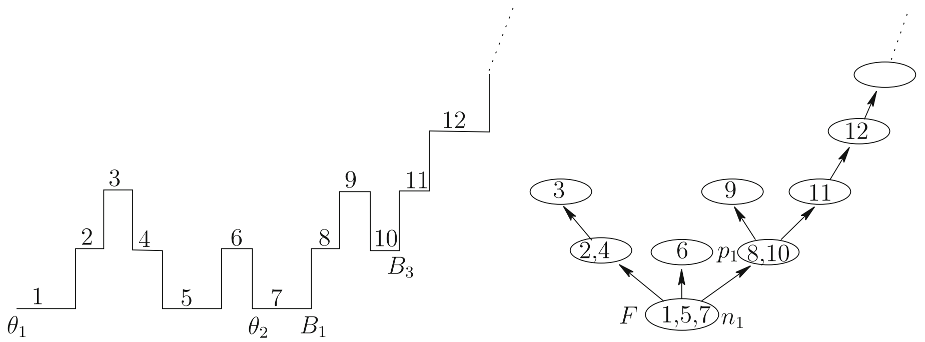

**Fig. 13** (*left*) A part of an infinite computation $\pi$ and (*right*) its corresponding stack-level tree $t_1$. The branch on the right is infinite. The notation correspond to Lemma 5

As in the case of Lemma 1, if the premises do not hold (i.e., the computations are not input-equivalent), Theorem 1 holds trivially. Also as in the case of Lemma 1, consequent 4 implies $\text{OutSeq}[\pi] = \text{OutSeq}[\tau]$ and hence the consequent of Theorem 1 for the case of infinite computations.

*Proof* (Lemma 5) By induction on the index of the nodes in the infinite branches. Let $B_1$ and $B_2$ be a pair of call configurations on the infinite branches of $t_1$ and $t_2$ respectively, which are at the same execution level, i.e., $B_1.d = B_2.d$.

*Base* Consider $n_1, n_2$, the root nodes of $t_1$ and $t_2$. For each of the branches originating from $n_1, n_2$ that are not on the infinite branches (these are nodes containing segments (2, 4) and (6) in $t_1$, as appears in the figure), Lemma 1 holds.

This allows us to use Lemma 4 (for example, between points $\theta_1$ and $\theta_2$ in the left drawing), and Lemma 2 (for example, between points $\theta_2$ and $B_1$ in the left drawing) with respect to subcomputations starting in the beginning of $F$ and $G$ and ending at $B_1, B_2$.

By consequent 3 of Lemma 2 $\langle p_1, p_2\rangle \in map$, which proves consequent 1 of the current lemma. Consequents 5,6 and 7 of Lemma 2 imply the other three consequents of the current lemma: $B_1.\sigma[\overline{e}_1] = B_2.\sigma[\overline{e}_2]$, $B_1.\overline{R} = B_2.\overline{R}$ and $B_1.\overline{W} = B_2.\overline{W}$.

*Step* Let the call configurations $B_3$ and $B_4$ be the successors of $B_1$ and $B_2$ respectively on the infinite branches. The subcomputations from $B_1$ to $B_3$ and from $B_2$ to $B_4$ are finite and therefore Lemmas 1–4 apply to them.

We now assume that the induction hypothesis holds for $B_1$ and $B_2$ and prove it for $B_3$ and $B_4$. By the induction hypothesis $B_1.\sigma[\overline{e}_1] = B_2.\sigma[\overline{e}_2]$, $B_1.\overline{R} = B_2.\overline{R}$, and $B_1.\overline{W} = B_2.\overline{W}$.

Let $n_3$ and $n_4$ be the nodes in the stack-level trees reached by the calls made at $B_1$ and $B_2$. For each of the branches originating from $n_3, n_4$ that are not on the infinite branches Lemma 1 holds.

Similarly to the base case, Lemmas 2 and 4 apply to the subcomputations starting in $B_1$ and $B_2$ and ending at $B_3$, $B_4$ respectively. By consequent 3 of Lemma 2 the procedures called at $B_3$ and $B_4$ are mapped to one another, which proves consequent 1 of the current lemma. Consequents 5,6 and 7 of Lemma 2 imply the other three consequents of the current lemma: $B_3.\sigma[\overline{e}_1] = B_4.\sigma[\overline{e}_2]$, $B_3.\overline{R} = B_4.\overline{R}$ and $B_3.\overline{W} = B_4.\overline{W}$.

(End of proof of Lemma 5) □

We proved for both finite and infinite computations that the premises of Theorem 3 imply its consequent. This concludes the proof of soundness for the (REACT- EQ) rule.

(End of proof of Theorem 3). □

## 5.4 Using rule (REACT- EQ): a long example

Every reactive program has at least one loop or recursion, and, recall, the former can be translated into recursion as well.

In this section we present an example of a pair of reactive programs which behave as a simple calculator over natural numbers. The calculated expressions can contain the "+" and "*" operations and the use of "(" and ")" to associate operations. The calculator obeys the operator precedence of "+" and "*". We set the domain $\mathbb{D} \doteq \mathbb{N} \cup \{``+", ``*", ``(", ``)"\}$, where "+","*","(" and ")" are constant symbols, and the constants TRUE and FALSE to be the 1 and 0 values respectively. We define + and * to be operators over $\mathbb{D}$. If $t_1, t_2 \in \mathbb{N}$ then the value of $t_1 + t_2$ and $t_1 * t_2$ is as given by the natural interpretation of this operations over $\mathbb{N}$. If $t_1 \notin \mathbb{N}$ or $t_2 \notin \mathbb{N}$ then we define $t_1 + t_2 = t_1 * t_2 = 0$. We assume also the existence of the equality operator = over $\mathbb{D}$.

The two programs in Fig. 14 are executed by a call to their respective "sum" procedures with 0 as the input argument. We assume that the input sequence to the program is a valid arithmetical expression. Each time a program reads ")" from the input sequence, it prints the value of the expression between the parentheses that this input symbol closes. We proceed with a short explanation of the programs' operation.

The procedures $sum^{\mathrm{L}}$ and $sum^{\mathrm{R}}$ receive the value of the sum until now in the formal arguments $v^{\mathrm{L}}$ or $v^{\mathrm{R}}$ respectively, add to it the value of the next product that they receive in variable $r^{\mathrm{L}}$ or $b^{\mathrm{R}}$ (from calls to $prod^{\mathrm{L}}$ or $prod^{\mathrm{R}}$), and if the next symbol is ")" they output the sum and return it in variable $r^{\mathrm{L}}$ or $r^{\mathrm{R}}$ respectively. If the next symbol is "+" they recursively call $sum^{\mathrm{L}}$ or $sum^{\mathrm{R}}$ to continue the summation.

Similarly, the procedures $prod^{\mathrm{L}}$ and $prod^{\mathrm{R}}$ receive the value of the product up to now in formal argument $v^{\mathrm{L}}$ and $v^{\mathrm{R}}$, multiply it by the value of the next natural number or expression in parentheses (received from $num^{\mathrm{L}}$ or $num^{\mathrm{R}}$ in variable $r^{\mathrm{L}}$ or $d^{\mathrm{R}}$), and get the next symbol in $op^{\mathrm{L}}$ or $op^{\mathrm{R}}$. If the next symbol is "*" they recursively call $prod^{\mathrm{L}}$ or $prod^{\mathrm{R}}$ to continue calculating the product. If the next symbol is "+" or ")", they just return the product value (in $r^{\mathrm{L}}$ or $r^{\mathrm{R}}$).

The $num^{\mathrm{L}}$ and $num^{\mathrm{R}}$ procedure may read a number or a "(" symbol from the input sequence. In the former case, they just return this number through $i^{\mathrm{L}}$ or $n^{\mathrm{R}}$. In the latter case, they call $sum^{\mathrm{L}}$ and $sum^{\mathrm{R}}$ to calculate the value of the expression inside the parentheses and return the result through $i^{\mathrm{L}}$ or $n^{\mathrm{R}}$.

The $getop^{\mathrm{L}}$ and $getop^{\mathrm{R}}$ just read a single symbol from the input sequence (it can be "+", "*" or ")") and return it in $op^{\mathrm{L}}$ or $op^{\mathrm{R}}$, respectively.

We use the (REACT- EQ) rule to prove reactive equivalence of $sum^{\mathrm{L}}$ and $sum^{\mathrm{R}}$, under the assumption that they receive a valid arithmetical expression. We introduce four uninterpreted procedures $U_s$, $U_p$, $U_n$ and $U_g$ and set the mapping UP to satisfy $UP(sum^{\mathrm{L}}) = UP(sum^{\mathrm{R}}) = U_s$, $UP(prod^{\mathrm{L}}) = UP(prod^{\mathrm{R}}) = U_p$, $UP(num^{\mathrm{L}}) = UP(num^{\mathrm{R}}) = U_n$ and $UP(getop^{\mathrm{L}}) = UP(getop^{\mathrm{R}}) = U_g$.

In Fig. 15 we present the SSA form of the formulas that represent the possible computations of the isolated procedure bodies. Each of the procedures has at most a single **input** or **output** statement. We mark the single input value by $in_1$ or $in_2$, and the single output value by $out_1$ or $out_2$.

```
procedure $sum^{L}$(val $v^{L}$; ret $r^{L}$):
    call $prod^{L}(1; r^{L}, op^{L})$;
    $r^{L} := r^{L} + v^{L}$;
    if $op^{L}$=‘)’ then
        output($r^{L}$);
    fi
    if $op^{L}$=‘+’ then
        call $sum^{L}(r^{L};\ r^{L})$
    fi
    return

procedure $prod^{L}$(val $v^{L}$; ret $r^{L}$,
$op^{L}$):
    call $num^{L}(r^{L})$;
    $r^{L} := r^{L} * v^{L}$;
    call $getop^{L}(op^{L})$;
    if $op^{L}$ = ‘*’ then
        call $prod^{L}(r^{L};\ r^{L},\ op^{L})$
    fi
    return

procedure $num^{L}$(val; ret $i^{L}$):
    input($i^{L}$);
    if $i^{L}$=‘(’ then
        call $sum^{L}(0;\ i^{L})$
    fi
    return

procedure $getop^{L}$(val; ret $op^{L}$):
    input($op^{L}$);
    return
```

```
procedure $sum^{R}$(val $v^{R}$; ret $r^{R}$):
    call $prod^{R}(1; b^{R}, op^{R})$;
    if $op^{R}$ =‘+’ then
        call $sum^{R}(v^{R} + b^{R};\ r^{R})$
    else
        $r^{R} := v^{R} + b^{R}$;
    fi
    if $op^{R}$=‘)’ then
        output($v^{R} + b^{R}$);
    fi
    return

procedure $prod^{R}$(val $v^{R}$; ret $r^{R}$,
$op^{R}$):
    call $num^{R}(d^{R})$;
    call $getop^{R}(op^{R})$;
    if $op^{R}$=‘*’ then
        call $prod^{R}(v^{R} * d^{R};\ r^{R},\ op^{R})$
    else
        $r^{R} := v^{R} * d^{R}$
    fi
    return

procedure $num^{R}$(val; ret $n^{R}$):
    input($i^{R}$);
    if $i^{R}$=‘(’ then
        call $sum^{R}(0;\ n^{R})$
    else
        $n^{R} := i^{R}$
    fi
    return

procedure $getop^{R}$(val; ret $op^{R}$):
    input($op^{R}$);
    return
```

**Fig. 14** Two reactive calculator programs (labels were removed for better readability). The programs output a value every time they encounter the “)” symbol

– **Premise 9.2 (*return-values-equiv*).** Checking this premise involves checking all input-equivalent subcomputations through the isolated bodies of each pair of the related procedures. As each of these isolated bodies include at most a single **input** statement, a sequence of a single input is enough for each of these checks. We denote this single input by $R_0$. We check the following formulas to be valid:

$$\begin{array}{ll}(v^{L} = v^{R} \wedge T_{sum^{L}} \wedge T_{sum^{R}}) & \rightarrow r^{L} = r^{R} \\ (v^{L} = v^{R} \wedge T_{prod^{L}} \wedge T_{prod^{R}}) & \rightarrow r^{L} = r^{R} \wedge op^{L} = op^{R} \\ (in_1 = R_0 \wedge in_2 = R_0 \wedge T_{num^{L}} \wedge T_{num^{R}}) & \rightarrow i^{L} = n^{R} \\ (in_1 = R_0 \wedge in_2 = R_0 \wedge T_{getop^{L}} \wedge T_{getop^{R}}) & \rightarrow op^{L} = op^{R}.\end{array} \qquad (10)$$

– **Premise 9.3 (*input-suffix-equiv*).** Recall that calls to uninterpreted procedures consume no values of the input sequence. Thus, to verify the satisfaction of *input-suffix-equiv*, we only need to check that any two related procedures have the same number of **input** statements. In this way, when the configurations at the beginning of the two isolated procedure

$$\underbrace{\begin{pmatrix} v_0^{\mathrm{L}} = v^{\mathrm{L}} & \wedge \\ U_p(1; r_1^{\mathrm{L}}, op_1^{\mathrm{L}}) & \wedge \\ r_2^{\mathrm{L}} = r_1^{\mathrm{L}} + v_0^{\mathrm{L}} & \wedge \\ (op_1^{\mathrm{L}} = \text{`)'} \rightarrow out_1 = r_2^{\mathrm{L}}) & \wedge \\ (op_1^{\mathrm{L}} = \text{`+'} \rightarrow U_s(r_2^{\mathrm{L}}; r_3^{\mathrm{L}})) & \wedge \\ (op_1^{\mathrm{L}} \neq \text{`+'} \rightarrow r_3^{\mathrm{L}} = r_2^{\mathrm{L}}) & \wedge \\ r^{\mathrm{L}} = r_3^{\mathrm{L}} & \end{pmatrix}}_{\boldsymbol{T_{sum^{\mathrm{L}}}}} \quad \underbrace{\begin{pmatrix} v_0^{\mathrm{R}} = v^{\mathrm{R}} & \wedge \\ U_p(1; b_1^{\mathrm{R}}, op_1^{\mathrm{R}}) & \wedge \\ & \\ (op_1^{\mathrm{R}} = \text{`+'} \rightarrow U_s(v_0^{\mathrm{R}} + b_1^{\mathrm{R}}; r_1^{\mathrm{R}})) & \wedge \\ (op_1^{\mathrm{R}} \neq \text{`+'} \rightarrow r_1^{\mathrm{R}} = v_0^{\mathrm{R}} + b_1^{\mathrm{R}}) & \wedge \\ (op_1^{\mathrm{R}} = \text{`)'} \rightarrow out_2 = v_0^{\mathrm{R}} + b_1^{\mathrm{R}}) & \wedge \\ r^{\mathrm{R}} = r_1^{\mathrm{R}} & \end{pmatrix}}_{\boldsymbol{T_{sum^{\mathrm{R}}}}}$$

$$\underbrace{\begin{pmatrix} v_0^{\mathrm{L}} = v^{\mathrm{L}} & \wedge \\ U_n(r_1^{\mathrm{L}}) & \wedge \\ r_2^{\mathrm{L}} = r_1^{\mathrm{L}} * v_0^{\mathrm{L}} & \wedge \\ U_g(op_1^{\mathrm{L}}) & \wedge \\ (op_1^{\mathrm{L}} = \text{`*'} \rightarrow U_p(r_2^{\mathrm{L}}; r_3^{\mathrm{L}}, op_2^{\mathrm{L}})) & \wedge \\ (op_1^{\mathrm{L}} \neq \text{`*'} \rightarrow (r_3^{\mathrm{L}} = r_2^{\mathrm{L}} \wedge & \\ \qquad op_2^{\mathrm{L}} = op_1^{\mathrm{L}})) & \wedge \\ r^{\mathrm{L}} = r_3^{\mathrm{L}} \wedge op^{\mathrm{L}} = op_2^{\mathrm{L}} & \end{pmatrix}}_{\boldsymbol{T_{prod^{\mathrm{L}}}}} \quad \underbrace{\begin{pmatrix} v_0^{\mathrm{R}} = v^{\mathrm{R}} & \wedge \\ U_n(d_1^{\mathrm{R}}) & \wedge \\ U_g(op_1^{\mathrm{R}}) & \wedge \\ (op_1^{\mathrm{R}} = \text{`*'} \rightarrow U_p(v_0^{\mathrm{R}} * d_1^{\mathrm{R}}; & \\ \qquad r_1^{\mathrm{R}}, op_2^{\mathrm{R}})) & \wedge \\ (op_1^{\mathrm{R}} \neq \text{`*'} \rightarrow (r_1^{\mathrm{R}} = v_0^{\mathrm{R}} * d_1^{\mathrm{R}} \wedge & \\ \qquad op_2^{\mathrm{R}} = op_1^{\mathrm{R}})) & \wedge \\ r^{\mathrm{R}} = r_1^{\mathrm{R}} \wedge op^{\mathrm{R}} = op_2^{\mathrm{R}} & \end{pmatrix}}_{\boldsymbol{T_{prod^{\mathrm{R}}}}}$$

$$\underbrace{\begin{pmatrix} i_0^{\mathrm{L}} = in_1 & \wedge \\ (i_0^{\mathrm{L}} = \text{`('} \rightarrow U_s(0; i_1^{\mathrm{L}})) & \wedge \\ (i_0^{\mathrm{L}} \neq \text{`('} \rightarrow i_1^{\mathrm{L}} = i_0^{\mathrm{L}}) & \wedge \\ i^{\mathrm{L}} = i_1^{\mathrm{L}} & \end{pmatrix}}_{\boldsymbol{T_{num^{\mathrm{L}}}}} \quad \underbrace{\begin{pmatrix} i_0^{\mathrm{R}} = in_2 & \wedge \\ (i_0^{\mathrm{R}} = \text{`('} \rightarrow U_s(0; n_1^{\mathrm{R}})) & \wedge \\ (i_0^{\mathrm{R}} \neq \text{`('} \rightarrow n_1^{\mathrm{R}} = i_0^{\mathrm{R}}) & \wedge \\ n^{\mathrm{R}} = n_1^{\mathrm{R}} & \end{pmatrix}}_{\boldsymbol{T_{num^{\mathrm{R}}}}}$$

$$\underbrace{\begin{pmatrix} op_0^{\mathrm{L}} = in_1 & \wedge \\ op^{\mathrm{L}} = op_0^{\mathrm{L}} & \end{pmatrix}}_{\boldsymbol{T_{getop^{\mathrm{L}}}}} \quad \underbrace{\begin{pmatrix} op_0^{\mathrm{R}} = in_2 & \wedge \\ op^{\mathrm{R}} = op_0^{\mathrm{R}} & \end{pmatrix}}_{\boldsymbol{T_{getop^{\mathrm{R}}}}}$$

**Fig. 15** The SSA corresponding to the programs in Fig. 14

bodies have equal input sequences, also the configurations at the end of these bodies have equal input sequences as the same prefix was consumed by the computations through the bodies. This condition is satisfied trivially for all mapped procedures.

– **Premise 9.4** (*call-output-seq-equiv*)**.** In procedures $sum^{\mathrm{L}}$, $sum^{\mathrm{R}}$, $prod^{\mathrm{L}}$, $prod^{\mathrm{R}}$, $num^{\mathrm{L}}$ and $num^{\mathrm{R}}$, the execution may take several paths. We need to compare the guard and input values of each procedure call and each **output** statement in each path. Note that the calls to $U_n$ and $U_g$ are always unconditioned and have no read arguments. Therefore, they trivially satisfy the *call-output-seq-equiv* conditions. The check involves validating the following formulas:

$$\begin{array}{l} (v^{\mathrm{L}} = v^{\mathrm{R}} \wedge T_{sum^{\mathrm{L}}} \wedge T_{sum^{\mathrm{R}}}) \rightarrow \\ ((1 = 1) \wedge \\ (op_1^{\mathrm{L}} = \text{`)'} \leftrightarrow op_1^{\mathrm{R}} = \text{`)'}) \wedge (op_1^{\mathrm{L}} = \text{`)'} \rightarrow out_1 = out_2) \wedge \\ (op_1^{\mathrm{L}} = \text{`+'} \leftrightarrow op_1^{\mathrm{R}} = \text{`+'}) \wedge (op_1^{\mathrm{L}} = \text{`+'} \rightarrow r_2^{\mathrm{L}} = v_0^{\mathrm{R}} + b_1^{\mathrm{R}})). \end{array} \tag{11}$$

It is easier to follow this formula while referring to the definition of $T_{sum^{\mathrm{L}}}$ and $T_{sum^{\mathrm{R}}}$. The second line asserts that the input arguments of $U_p$ are the same. The third line asserts that the guards of the **output** statements are the same, and if they both hold, then the output value is the same. The last line asserts that the guards of the call to $U_s$ are the same, and if they both hold, then the read arguments of $U_s$ are the same.

$$\begin{aligned}&(v^{\text{L}} = v^{\text{R}} \wedge T_{prod^{\text{L}}} \wedge T_{prod^{\text{R}}}) \rightarrow \\ &((op_1^{\text{L}} = \text{`*'} \leftrightarrow op_1^{\text{R}} = \text{`*'}) \wedge (op_1^{\text{L}} = \text{`*'} \rightarrow r_2^{\text{L}} = v_0^{\text{R}} * d_1^{\text{R}}))\end{aligned} \tag{12}$$

$$\begin{aligned}&(in_1 = R_0 \wedge in_2 = R_0 \wedge T_{num^{\text{L}}} \wedge T_{num^{\text{R}}}) \rightarrow \\ &((i_0^{\text{L}} = \text{`('} \leftrightarrow i_0^{\text{R}} = \text{`('}) \wedge (i_0^{\text{L}} = \text{`('} \rightarrow 0 = 0)).\end{aligned} \tag{13}$$

Using the uninterpreted procedure relations and commutativity of the "+" and "*" operators, one can prove the validity of (10)–(13). Note that uninterpreted procedures which have no read arguments return non-deterministic but constant values in their write arguments.

This concludes the verification of the premises of rule (REACT- EQ), which establishes, among other things, that $reactive\text{-}equiv(sum^{\text{L}},\ sum^{\text{R}})$ holds. Consequently we know that the two programs generate the same output sequence when executed on the same arithmetical expression.

## 6 What the rules cannot prove

All three rules rely on a 1-1 and onto mapping of the procedures (possibly after inlining of some of them, as mentioned in the introduction), such that every pair of mapped procedures are computationally equivalent. Various semantic-preserving code transformations do not satisfy this requirement. Here are a few examples:

1. Consider the following equivalent procedures, which, for a natural number $n$, compute $\sum_{i=1}^{n} i$.

```
procedure F(val $n$; ret $r$):
    if $n \leq 1$ then $r := n$
    else
        call $F(n-1, r)$;
        $r := n + r$
    fi
    return
```

```
procedure G(val $n$; ret $r$):
    if $n \leq 1$ then $r := n$
    else
        call $G(n-2, r)$;
        $r := n + (n-1) + r$
    fi
    return
```

   Since the two procedures are called with different arguments, their computational equivalence cannot be proven with rule (PROC- P- EQ).

2. Consider a similar pair of equivalent procedures, that this time make recursive calls with equivalent arguments:

```
procedure F(val $n$; ret $r$):
    if $n \leq 0$ then $r := n$
    else
        call $F(n-1, r)$;
        $r := n + r$
    fi
    return
```

```
procedure G(val $n$; ret $r$):
    if $n \leq 1$ then $r := n$
    else
        call $G(n-1, r)$;
        $r := n + r$
    fi
    return
```

   The premise of (PROC- P- EQ) fails due to the case of $n == 1$.

3. We now consider an example in which the two programs are both computational equivalent and reach-equivalent, but still our rules fail to prove it.

```
procedure F(val $n$; ret $r$):
    if $n \leq 0$ then $r := 0$
    else
        call $F(n-1, r)$;
        $r := n + r$
    fi;
    return
```

```
procedure F(val $n$; ret $r$):
    if $n \leq 0$ then $r := 0$
    else
        call $F(n-1, r)$;
        if $r \geq 0$ then $r := n + r$;
        fi
    fi
    return
```

In this case the "if" condition in the second program always holds. Yet since the Uninterpreted Functions return arbitrary, although equal, values, they can return a negative value, which will make this "if" condition not hold and as a result make the two isolated functions return different values.

## 7 Summary

We presented three proof rules in the style of Hoare's rule for recursive procedures: rule (PROC- P- EQ) proves partial equivalence between programs, rule (M- TERM) proves mutual termination of such programs and, finally, rule (REACT- EQ) proves reactive-equivalence between reactive programs, and also generalizes the first two.

These rules can be used in any one of the scenarios described in the introduction. We are using them as part of an automated regression-verification tool for C programs that we currently develop, and so far used them for proving such equivalence of several non-trivial programs. Deriving the proper verification conditions automatically is easy in the isolated procedures, and the verification conditions themselves are typically not hard to solve with the underlying proof engine that we use, namely CBMC.[8] Once this system will be capable of handling a larger set of real programs (it currently does not support various language features of C), it will be interesting to see in how many cases real changes, made between versions of real programs, can be proven to be equal with the rules described in this article.

## Appendix A: Formal definitions for Sect. 5

**Definition 19 (Input/output subsequences of a subcomputation)**
Let $\pi'$ be a subcomputation of a computation $\pi$. If $\pi'$ is finite then the *input subsequence* of $\pi'$ is the prefix sequence $Q_{IN}$ that satisfies first$[\pi'].\overline{R} = Q_{IN} \cdot$ last$[\pi'].\overline{R}$ and the *output subsequence* of $\pi'$ is the tail sequence $Q_{OUT}$ that satisfies first$[\pi'].\overline{W} \cdot Q_{OUT} =$ last$[\pi'].\overline{W}$. If $\pi'$ is infinite then the *input subsequence* of $\pi'$ is simply first$[\pi'].\overline{R}$ and the *output subsequence* of $\pi'$ is the tail sequence $Q_{OUT}$ that satisfies first$[\pi'].\overline{W} \cdot Q_{OUT} =$ OutSeq$[\pi]$. □

Less formally, an input subsequence of a subcomputation $\pi'$ is the subsequence of inputs that is "consumed" by $\pi'$, whereas the output subsequence of a subcomputation $\pi'$ is the subsequence of outputs of $\pi'$.

We mark the input subsequence of $\pi'$ by $\Delta\overline{R}[\pi']$ and its output subsequence by $\Delta\overline{W}[\pi']$.

In all subsequent definitions $P_1$ and $P_2$ are LPL+IO programs.

[8] This depends more on the type of operators there are in the procedures than the sheer size of the program. For example, a short program that includes a multiplication between two integers is hard to reason about regardless of the length of the procedure.

**Definition 20 (Reactive equivalence of two procedures)**
Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map$, if for every two subcomputations $\pi'_1$ and $\pi'_2$ that are input equivalent with respect to $F$ and $G$ it holds that $\Delta\overline{W}[\pi'_1] = \Delta\overline{W}[\pi'_2]$ then $F$ and $G$ are *reactively equivalent.* □

Denote by $reactive\text{-}equiv(F, G)$ the fact that $F$ and $G$ are reactively equivalent.

**Definition 21 (Return-values equivalence of two reactive procedures)**
If for every two finite subcomputations $\pi'_1$ and $\pi'_2$ that are input equivalent with respect to procedures $F$ and $G$ it holds that

$$\text{last}[\pi'_1].\sigma[\,\overline{arg\text{-}w_F}\,] = \text{last}[\pi'_2].\sigma[\,\overline{arg\text{-}w_G}\,]$$

then $F$ and $G$ are *Return-values equivalent.* □

(Recall that input-equivalent subcomputations are also argument-equivalent and hence maximal—see Definition 9).

Denote by $return\text{-}values\text{-}equiv(F, G)$ the fact that $F$ and $G$ are return-value equivalent.

**Definition 22 (Inputs-suffix equivalence of two reactive procedures)**
If for every two finite subcomputations $\pi'_1$ and $\pi'_2$ that are input equivalent with respect to procedures $F$ and $G$ it holds that

$$\Delta\overline{R}[\pi'_1] = \Delta\overline{R}[\pi'_2],$$

then $F$ and $G$ are *Inputs-suffix equivalent.* □

Denote by *input-suffix-equiv*$(F, G)$ the fact that $F$ and $G$ are inputs-suffix equivalent.

**Definition 23 (Output configuration)** A configuration $C$ is an *output configuration* if current-label$[C]$ = before[**output**$(e)$]. □

**Definition 24 (Call and output sequence of a subcomputation)** The *call and output sequence* of a subcomputation $\pi'$ contains all the call and output configurations in $\pi'$ in the order in which they appear in $\pi'$. □

**Definition 25 (Call and output sequence equivalence between subcomputations)** Finite subcomputation $\pi'_1$ and $\pi'_2$ from some levels are *call and output sequence equivalent* if the call-and-output-sequences $CC_1$ of $\pi'_1$ and $CC_2$ of $\pi'_2$, satisfy:

1. $|CC_1| = |CC_2|$
2. If for some $i \in \{1, \ldots, |CC_1|\}$, current-label$[(CC_1)_i]$ = before[**call** $p_1(\overline{e}_1; \overline{x}_1)$]), then
   - current-label$[(CC_2)_i]$ = before[**call** $p_2(\overline{e}_2; \overline{x}_2)$],
   - $\langle p_1, p_2\rangle \in map$, and
   - $(CC_1)_i.\sigma[\,\overline{e_1}\,] = (CC_2)_i.\sigma[\,\overline{e_2}\,]$.
3. If for some $i \in \{1, \ldots, |CC_1|\}$) current-label$[(CC_1)_i]$ = before[**output**$(e_1)$]), then
   - current-label$[(CC_2)_i]$ = before[**output**$(e_2)$], and
   - $(CC_1)_i.\sigma[e_1] = (CC_2)_i.\sigma[e_2]$. □

Extending this definition to procedures, we have:

**Definition 26 (Call and output sequence equivalence of two procedures)** Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map$, if for every two finite subcomputations $\pi_1'$ and $\pi_2'$ that are input equivalent with respect to $F$ and $G$ it holds that $\pi_1'$ and $\pi_2'$ are call and output sequence equivalent, then $F$ and $G$ are *call and output sequence equivalent*. □

Denote by $call\text{-}output\text{-}seq\text{-}equiv(F, G)$ the fact that $F$ and $G$ are call-sequence equivalent.

## Appendix B: Refactoring rules that our rules can handle

It is beneficial to categorize refactoring rules that can be handled by our proof rules. In general, every change that is local to the procedure, and does not move code between different iterations of a loop or recursion, can be handled by the proof rules.

Considering the list of popular refactoring rules in [4] (while ignoring those that are specific to object-oriented code, or Java):

- Our rules can handle the following rules: *Consolidate Duplicate Conditional Fragments, Introduce Explaining Variable, Reduce Scope of Variable, Remove Assignments to Parameters, Remove Control Flag, Remove Double Negative, Replace Assignment with Initialization, Replace Iteration with Recursion, Replace Magic Number with Symbolic Constant, Replace Nested Conditional with Guard Clauses, Replace Recursion with Iteration, Reverse Conditional, Split Temporary Variable, Substitute Algorithm.*
- If the rules are used in a decision procedure that is able to inline code, they can also prove the correctness of the following refactoring rules:
  *Decompose Conditional, Extract Method, Inline Method, Inline Temp, Replace Parameter with Explicit Methods, Replace Parameter with Method, Replace Temp with Query, Self Encapsulate Field, Separate Data Access Code.*
- Finally, the following refactoring rules cannot be handled by our rules:
  *Replace Static Variable with Parameter* (change in the prototype of the function), *Separate Query from Modifier* (splits a function to two functions with different behaviors), *Split Loop* (since in our setting, loops are modeled as recursive functions, this transformation turns one recursive function into two).

## References

1. Arons, T., Elster, E., Fix, L., Mador-Haim, S., Mishaeli, M., Shalev, J., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zuck, L.D.: Formal verification of backward compatibility of microcode. In: Etessami, K., Rajamani, S. (eds.) Proceedings of 17th International Conference on Computer Aided Verification (CAV'05), Lecture Notes in Computer Science, vol. 3576. Springer, Edinburgh (2005)
2. Bouge, L., Cachera, D.: A logical framework to prove Properties of alpha programs (revised version). Tech. Rep. RR-3177 (1997). citeseer.ist.psu.edu/bouge97logical.html
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst **13**(4), 451–490 (1991)
4. Fowler, M.: http://www.refactoring.com
5. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley, Menlo Park (1999)
6. Francez, N.: Program Verification. Addison-Wesley, Wokingham (1993)

7. Hoare, C.: Prcedures and parameters: an axiomatic approach. In: Proceedings of Symposium on Semantics of Algorithmic Languages, vol. 188, pp.102–116. Springer, New York (1971)
8. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of DAC 2003, pp. 368–371. ACM Press, New York (2003)
9. Luckham, D., Park, D., Paterson, M.: On formalized computer programs. J. Comput. Syst. Sci. **4**(3), 220–249 (1970)
10. Manolios, P.: Computer-Aided Reasoning: ACL2 Case Studies, Chap. Mu-Calculus Model-Checking, pp. 93–111. Kluwer Academic Publishers, Dordrecht (2000)
11. Manolios, P., Kaufmann, M.: Adding a total order to acl2. In: The Third International Workshop on the ACL2 Theorem Prover (2002)
12. Manolios, P., Vroon, D.: Ordinal arithmetic: algorithms and mechanization. J Autom Reason (2006) (to appear)
13. Pratt, T.W.: Kernel equivalence of programs and proving kernel equivalence and correctness by test cases. International Joint Conference on Artificial Intelligence (1971)
14. Shostak, R.: An algorithm for reasoning about equality. Commun ACM **21**(7), 583–585 (1978)