Regression Verification: Theoretical and Implementation Aspects

Benny Godlin

Regression Verification: Theoretical and Implementation Aspects

Research Thesis

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

Benny Godlin

Submitted to the Senate of
the Technion - Israel Institute of Technology

ADAR, 5768 HAIFA MARCH, 2008

The Research Thesis Was Done Under The Supervision of
Dr. Ofer Strichman in the Faculty of Computer Science.

The Generous Financial Help Of the Technion
Is Gratefully Acknowledged.

Table of Contents

List of Figures

Abstract

Proving the equivalence of successive (closely related) versions of a program has the potential of being easier in practice than functional verification, although both problems are undecidable. There are two main reasons for this claim: it circumvents the problem of specifying what the program should do, and in many cases it is computationally easier. In this thesis we study theoretical and practical aspects of this problem, which we call *regression verification*.

The thesis is divided into two parts. In the first part we propose several notions of equivalence between programs, and corresponding proof rules in the style of Hoare's rule for recursive procedures. These rules enable us to prove the equivalence of recursive and mutually recursive programs, and also have an advantage from the perspective of the computational effort, since it allows us to decompose and abstract the two programs. This method is sound but incomplete.

In the second part we describe a regression verification tool for C programs, based on the above-mentioned rules, that we built on top of a software bounded model-checker called CBMC.

1 General introduction

1.1 Regression vs. functional verification

In 2003 Tony Hoare declared a "grand challenge" to the computer science community to build a *verifying compiler*, i.e., a compiler that proves that the input program is functionally correct with respect to a given specification [20]. Quoting from a later publication of Hoare and Misra regarding the nature of this challenge [21], " ...*the time is ripe to embark on an international Grand Challenge project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs[...] the program verifier will be based on a sound and complete theory of programming; they will be supported by a range of program construction and analysis tools[...] The project will provide the scientific basis of a solution for many of the problems of programming error ...*".

Regardless of the difficulty of building such a compiler (the problem is obviously undecidable in general), a major problem in using it on a broad scale is the difficulty of specifying what the program should do. Large parts of realistic programs are often hard to specify beyond assertions of local properties. In many cases the process of describing what a code segment should do is as difficult and at least as complicated as the coding itself. Software testing is perhaps a good parallel which indicates what can be expected to succeed in industry: high-level temporal property-based testing, although by now supported by commercial tools such as Temporal-Rover[8], is of very limited use. Industry typically attempts to circumvent the specification problem with *Regression Testing*, which is probably the most popular testing method for general computer programs. It is based on the idea of reasoning by induction: check an initial version of the software when it is still very simple, and then check that a newer version of the software produces the same output as the earlier one, given the same inputs. If the result of this process is a counterexample, the user is asked to check whether it is an error or a legitimate change. In the latter case the testing database is updated with the new 'correct' output value. Regression Testing does not require a

formal specification of the investigated system nor a deep understanding of the code, which makes it highly suitable for accompanying the development process, especially if it involves more than one programmer.

In this thesis we develop techniques for *Regression Verification*, namely techniques for proving the equivalence of programs[1], with focus on the case of two closely related versions of the same code. The problem of program equivalence can be reduced to one of functional verification of a single program which merges the two compared programs, but this direct approach makes no use of the expected similarity of the code. Program equivalence is undecidable (see [35], which examines conditions under which the equivalence problem is undecidable) and no doubt a grand challenge in its own right, but it is expected to be easier than building a program verifier as envisioned by the grand challenge: First, there is no need for a formal specification, other than pointing out which expressions should be evaluated to the same value in the two versions of the program; Second, as we show in this thesis, there are various abstraction and decomposition techniques that are applicable to equivalence checking but not to functional verification[2]; Third, there are techniques that can make the computation easier if a large part of the code has not changed between the two compared versions.

What motivated us to study the general problem of equivalence, is that only in the last few years tools for functional verification of realistic programs have been developed and made available (tools such as SLAM [2],Blast [18], Magic [4] and CBMC [22], to name a few). Such tools can serve as the underlying engine of a regression verification tool. Indeed in this work we generate verification conditions in the form of small C programs, and then discharge them with a C bounded model checker called CBMC [22].

[1]There are different ways to define program equivalence. We dedicate Sect. 1.4 to this question.

[2]The same observation is well known in the hardware domain, where *equivalence checking* of circuits is considered computationally easier in practice than model-checking.

The tool CBMC. CBMC, developed by D. Kroening, supports most of the features of ANSI-C. It requires from the user to define a bound k on the number of iterations that each loop in a given ANSI-C program is taken, and a similar bound on the depth of each recursion. 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. Proving such k-equivalence between programs is all that we need in our default setting, as our verification conditions are in the form of programs without loops or recursive calls.

1.2 When can regression verification be useful?

A natural question to ask is whether proving equivalence is relevant in the context of a real software development process, as in such a process the program is expected to produce a different output after every revision. While this is in general true, consider the following scenarios, all of which are targeted by our approach:

- Checking side-effects of new code. Suppose, for example, that from version 1.0 to version 1.1 a new flag was added, that 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. The regression verification tool that we developed in this thesis allows the user to express a condition (the inactivation of the flag in this case) under which the two programs are expected to produce equal outputs.

- Checking performance optimizations. After adding an optimization of the code for performance purposes, it is desirable to verify that the two versions of the code still produce the same output.

- Manual *Re-factoring*: Refactoring is a popular set of techniques for rewriting existing code for various purposes. To quote Martin Fowler [13, 12], the founder

```
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);
}
```

Figure 1: A refactoring example.

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.*' Proof of the equivalence of the code before and after refactoring, seems valuable in this case.

The following example demonstrates the need for proving equivalence of recursive functions after an application of a single refactoring rule.

Example 1. *The two equivalent programs in Figure 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 (i.e., they both either terminate or not), can be proved with the rules introduced in this thesis.*

□

A list of some of the refactoring rules that can be handled by our proposed rules is given in Appendix B.

1.3 Related work

The idea of proving equivalence between programs is not new, and in fact preceded the idea of functional verification.[3] It is a rather old challenge in the theorem-proving community (some recent examples include [3, 5, 24, 25, 26]). We are not aware of such works that are targeted at programs that are mostly equal, which is the target of regression verification, or of full mechanization. The theorem-proving works that we have seen (including those cited here that are based on ACL2) are mostly concerned with program equivalence as a case study for using proof techniques that are generic (i.e., not specific for proving equivalence).

Attempts to build fully automatic proof engines for industrial programs concentrated so far, to the best of our knowledge, on very restricted cases. Feng and Hu considered the problem of proving equivalence of embedded code [9, 10]. The main technique used in this line of work (also see [6]) is to prove the equivalence of small segments of the code which are loop-free. In such verification tasks methods that were developed for equivalence checking of combinatorial circuits become relevant (e.g., inserting cut-points in the case of [10]). Arons et al. [1] developed a tool in Intel for proving the equivalence of two versions of microcode, with the goal of proving backwards compatibility. The idea there is to compute symbolically the result of executing the code in both programs (which is assumed to be loop free). The verification condition is then a conjunction of these two symbolic paths and a condition that enforces the equality between their respective results. Both cases, then, give solution only to loop-free code.

Another relevant line of research is concerned with *translation validation* [30, 28, 29, 27, 36, 17, 32], the process of proving equivalence between a source and a target of

[3] In his 1969 paper about axiomatic basis for computer programming [19], Hoare points to previous works from the late 50's on axiomatic treatment of the problem of proving equivalence between programs.

a compiler or a code generator. The fact that the translation is mechanical allows the verification methodology to rely on various patterns and restrictions on the generated code. For example, translation validation for synchronous languages [30, 28, 29] relies on the fact that the target C code is loop-free. The translation validation tool from SDL to C by Haroud and Biere [17] is based on a variation of Floyd's method [11] for proving equivalence: it declares cutpoints in both programs (as in the original Floyd's method, there should be at least one cutpoint in each loop), maps them between the two programs, and proves that two related cutpoints are equivalent with respect to the 'observable' variables if they are equivalent in the preceding pair of cutpoints. This method works in the boundary of a single function on each side and does not support general programs (e.g. recursive programs). Yet another line of research is proving the equivalence of a C program and its realization in a hardware description language such as Verilog [22]. The Verilog design and the C program are both 'unrolled' k times and compared with a bounded model-checker. Thus, the two sides can be proven equal only up to a given depth (this is what we call later on k-equivalence).

1.4 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.

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 [3].
- The definitions of 'strong equivalence' and 'functional equivalence' in [23] and [33], 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.

Our tool attempts to prove equivalence following the first four definitions.

1.5 The structure of the thesis

In Part I of the thesis (Sects. 2– 7) we describe three proof rules for checking the first three notions of equivalence as described above, and prove their soundness.[4] In Part II of the thesis (Sects. 9 – 10) we describe the regression verification tool (RVT) that we have developed for checking these rules in the context of C programs. Using this tool we were able to prove the equivalence of several programs, which we list in Sect. 9.5. We dedicate Sect. 10 to discussing future work and what still has to be implemented in the tool in order for it to scale better and cover a larger set of realistic examples.

A word of warning: there is a certain gap between the first and second part that is related to the programming language: whereas the first part is described with a simple programming language that we call Linear Programming-Language (LPL), the second part is described in the context of C. In Sect. 9.1 we describe the restrictions on the input C programs that are required in order to apply the method suggested in the first part.

Another possible source of confusion is the use of the term 'procedure' and 'function'. In the first part of the thesis we define the rules over procedures, so more than one output is possible. We use 'function' in its mathematical sense when referring to uninterpreted functions. And, finally, in the second part, we use the common term of 'C functions', but in fact those are procedures, as they do not necessarily terminate nor necessarily return a value.

[4] These sections are taken almost verbatim from [15]

Part I

Inference Rules for Proving the Equivalence of Recursive Procedures

2 Overview of the three rules

In this section we give an informal description of the rules. Sections 5 – 7 describe these rules formally and prove their soundness. Let P_1 and P_2 be the two programs that we wish to prove equivalent. Assume that there is a one-to-one mapping between the procedures of P_1 and P_2 such that mapped procedures have the same prototype.[5] 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.

[5] 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 for the first part of the thesis, there is only one type and hence prototypes can be characterized by the number of arguments.

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 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. Assume that all loop constructs such as "while" and "for" statements are converted to recursion (we describe this transformation later in Appendix C.1). The rules 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 [22], which translates code with a bounded number of loops and recursive calls (in our case, none) to a propositional formula.

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 Section 4, together with its operational semantics. In Sections 5, 6 and 7 we state the three inference rules respectively and prove their soundness. Each rule is accompanied with an example.

3 Preliminaries

3.1 Notation

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.

3.2 Uninterpreted functions

Much of the work done in this thesis is based on the notion of *uninterpreted functions*. Uninterpreted functions and their natural extension to uninterpreted procedures, are useful for abstracting functions in verification conditions. No axiom is associated with them other than the congruence axiom, which enforces functional consistency (i.e., for equal inputs, the function returns an equal value):

$$\frac{x_1 = y_1 \wedge \cdots \wedge x_n = y_n}{F(x_1, \ldots, x_n) = F(y_1, \ldots, y_n)} \quad \text{(FUNCTIONAL CONGRUENCE)}\,, \tag{3.1}$$

where F is an arbitrary function.

4 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 [14], 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):

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

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 $\mathbb{Z} \cup \{\text{TRUE}, \text{FALSE}\}$ 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
```

*Figure 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 $PC \doteq \bigcup_{p \in Proc} PC_p$.

4.1 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[6] such that $pc_0 \in$

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

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 σ 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:

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

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 σ, 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 σ 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 σ *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$[S_{p_0}]$ and $d = 0$ is called the *terminal* configuration. $\diamond$

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:

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

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

   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

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

   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.

4.2 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 $C_i \rightarrow C_{i+1}$. If the computation is finite then the last configuration must be terminal.

The proofs throughout this thesis 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.

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

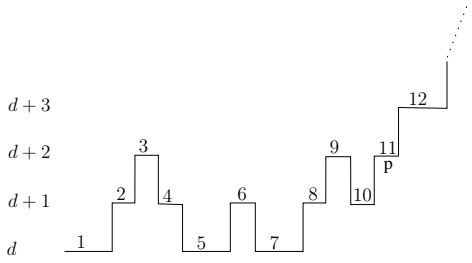

Figure 3: A computation through various stack levels. Each rise corresponds to a procedure call, and each fall to a **return** *statement.*

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

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

4.3 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 $map_f : Proc[P_1] \mapsto Proc[P_2]$ such that if $\langle F, G \rangle \in map_f$ 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.

5 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 Sections 6 and 7.

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.

```
procedure gcd1(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        l1  call gcd1(b, a; g)  l3
    fi;
    return
```

```
procedure gcd2(val x,y; ret z):
    z := x;
    if y > 0 then
        l2  call gcd2(y, z mod y; z)
l4
    fi;
    return
```

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

5.1 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 $\langle F, G\rangle \in map_f$, for any two computations π_1 in P_1 and π_2 in P_2, π_1' and π_2' are *argument-equivalent with respect to F and G* if the following holds:

1. π_1' and π_2' are maximal subcomputations of π_1 and π_2 from some levels d_1 and d_2 respectively,
2. current-label[first[π_1']] = before[F] and current-label[first[π_2']] = before[G], and
3. first[π_1'].$\sigma[\overline{arg\text{-}r_F}]$ = first[π_2'].$\sigma[\overline{arg\text{-}r_G}]$,

$\diamond$

We remind that global variables and static variables are not permitted in LPL directly, assuming that those can always be sent as input arguments to the procedures.

Definition 10 (Partial computational equivalence of procedures)**.** If for every argument-equivalent finite subcomputations π'_1 and π'_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.* $\diamond$

Denote by *comp-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 π_1 and π_2 through U,

$$\begin{aligned} &\text{first}[\pi_1].\sigma[\overline{arg\text{-}r_U}] = \text{first}[\pi_2].\sigma[\overline{arg\text{-}r_U}] \\ &\rightarrow \\ &\text{last}[\pi_1].\sigma[\overline{arg\text{-}w_U}] = \text{last}[\pi_2].\sigma[\overline{arg\text{-}w_U}] \ . \end{aligned} \tag{5.1}$$

There are well known decision procedures for reasoning about formulas that involve uninterpreted *functions* – see, for example, Shostak's algorithm [34], and accordingly most theorem provers support them. Such algorithms can be easily adapted to handle procedures rather than functions.

$$\frac{\begin{array}{ll}(5.3.1) & \forall \langle F, G\rangle \in map_f.\ \{ \\ (5.3.2) & \qquad \vdash_{\mathbb{L}_{\mathrm{UP}}}\ comp\text{-}equiv(F^{UP}, G^{UP})\ \ \} \\ \hline (5.3.3) & \qquad \forall \langle F, G\rangle \in map_f.\ comp\text{-}equiv(F, G)\end{array}}{\text{(PROC-P-EQ)}} \tag{5.3}$$

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

5.2 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:

$$\langle F, G\rangle \in map_f \iff \mathrm{UP}(F) = \mathrm{UP}(G)\ , \tag{5.2}$$

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., $F^{UP} \doteq F[f \leftarrow \mathrm{UP}(f) | f \in Proc[P]]$. ◇

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 $\langle F, G\rangle \in map_f$. 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, and hence, in particular, no recursion[7], 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 $map_f = \{\langle f, g\rangle\}$.*

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

  Recall that $f[f \leftarrow UP(f)]$ means that the call to f inside f is replaced with a call to UP(f) (isolation). Also recall that by definition of map_f (see (5.2)), $UP(f) = UP(g)$.

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

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

□

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., $H = UP(gcd_1) = UP(gcd_2)$. Figure 6 presents the isolated programs.*

[7] 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.

```
procedure gcd1(val a,b; ret g):
    if b = 0 then
        g := a
    else
        a := a mod b;
        call H(b, a; g)
    fi;
    return
```

```
procedure gcd2(val x,y; ret z):
    z := x;
    if y > 0 then
        call H(y, z mod y; z)
    fi;
    return
```

Figure 6: After isolation of the procedures, i.e., replacing their procedure calls with calls to the uninterpreted procedure H.

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) [7]. 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.*

The SSA form of gcd_1 is

$$T_{gcd_1} = \left(\begin{array}{ll} a_0 = a & \wedge \\ b_0 = b & \wedge \\ b_0 = 0 \rightarrow g_0 = a_0 & \wedge \\ (b_0 \neq 0 \rightarrow a_1 = (a_0 \mod b_0)) \wedge (b_0 = 0 \rightarrow a_1 = a_0) & \wedge \\ (b_0 \neq 0 \rightarrow H(b_0, a_1; g_1)) \wedge (b_0 = 0 \rightarrow g_1 = g_0) & \wedge \\ g = g_1 & \end{array} \right) . \qquad (5.4)$$

The SSA form of gcd_2 is

$$T_{gcd_2} = \left(\begin{array}{ll} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ (y_0 > 0 \rightarrow H(y_0, (z_0 \mod y_0); z_1)) \wedge (y_0 \leq 0 \rightarrow z_1 = z_0) & \wedge \\ z = z_1 & \end{array} \right) . \quad (5.5)$$

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:*

$$(a = x \wedge b = y \wedge T_{gcd_1} \wedge T_{gcd_2}) \quad \rightarrow \quad g = z \; . \quad (5.6)$$

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.

5.3 Rule (PROC-P-EQ) is sound

Let π be a computation of some program P_1. Each subcomputation π' *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 π'. 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 π 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 calling procedure in 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 π) 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 π' *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 π 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),*

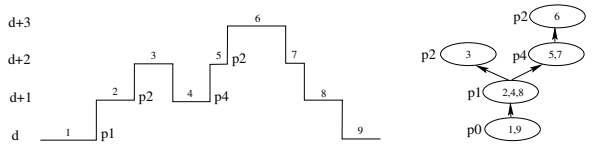

Figure 7: (*left*) A computation and its stack levels. The numbering on the horizontal segments are for reference only. (*right*) The stack-level tree corresponding to the computation on the left.

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, π_1 and π_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 $\langle F, G\rangle \in map_f$. Assume also that π_1 and π_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 π_1 and π_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 $\mathbb{L}_{\mathbb{UP}}$, π_1 and π_2 must satisfy *comp-equiv*(F^{UP}, G^{UP}) which entails the equality of $\overline{arg\text{-}w}$ values at their ends. Therefore, π_1 and π_2 satisfy the condition in *comp-equiv*(F, G).

Step: Assume the consequent (5.3.3) 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 $i+1$.

1. Consider the computation π_1. We construct a computation π_1' in F^{UP}, which is the same as π_1 in the level of n_1, with the following change. Each subcomputation of π_1 at 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 π_2' in G^{UP} corresponding to π_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 π_1 and π_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 $A_1'.\sigma[\overline{arg\text{-}r}_F] = A_2'.\sigma[\overline{arg\text{-}r}_G]$.

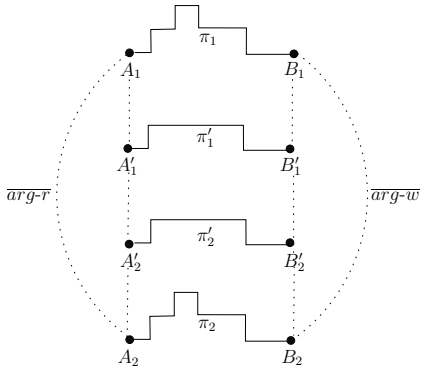

Figure 8: A diagram for the proof of Theorem 1. Dotted lines indicate an equivalence (either that we assume as a premise or that we need to prove) in the argument that labels the line. We do not write all labels to avoid congestion – see more details in the proof. π_1 is a subcomputation through F. π'_1 is the the corresponding subcomputation through F^{UP}, the isolated version of F. The same applies to π_2 and π'_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 $B.2$.

3. We now prove that the subcomputations π'_1 and π'_2 are valid computations through F^{UP} and G^{UP}. As π'_1 and π'_2 differ from π_1 and π_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 (5.1). Other parts of π'_1 and π'_2 are valid because π_1 and π_2 are valid subcomputations. Consider any pair of calls c_1 and c_2 in π_1 and π_2 from the current levels $d[n_1]$ and $d[n_2]$ to procedures p_1 and p_2 respectively, such that $\langle p_1, p_2\rangle \in map_f$. Let c'_1 and c'_2 be the calls to $\mathrm{UP}(p_1)$ and $\mathrm{UP}(p_2)$ which replace c_1 and c_2 in π'_1 and π'_2. Note that $\mathrm{UP}(p_1) = \mathrm{UP}(p_2)$ since $\langle p_1, p_2\rangle \in map_f$.

   By the induction hypothesis, procedures p_1, p_2 satisfy *comp-equiv*(p_1, p_2) for all subcomputations of depth $\leq i$, and in particular for subcomputations of π_1, π_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, π'_1 and π'_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-equiv*(F^{UP}, G^{UP}). Particularly, as π'_1 and π'_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 π_1 and π_2 satisfy *comp-equiv*(F, G).

□

6 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.

6.1 Definitions

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

Denote by *mutual-terminate*(F, G) the fact that F and G are mutually terminating.

$$\frac{\begin{array}{ll}(6.1.1) & \forall\langle F,G\rangle \in map_f.\ \{ \\ (6.1.2) & \quad comp\text{-}equiv(F,G) \wedge \\ (6.1.3) & \quad \vdash_{\mathbb{L}_{\mathbb{UP}}} \ reach\text{-}equiv(F^{UP},G^{UP})\ \}\end{array}}{(6.1.4) \quad \forall\langle F,G\rangle \in map_f.\ mutual\text{-}terminate(F,G)}\ (\textsc{m-term}) \tag{6.1}$$

Figure 9: Rule (M-TERM): An inference rule for proving the mutual termination of procedures. Note that Premise 6.1.2 can be proven by the (PROC-P-EQ) rule.

Definition 14 (Reach equivalence of procedures)**.** Procedures F and G are *reach-equivalent* if for every pair of argument-equivalent subcomputation π and τ 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 $\langle p_1, p_2\rangle \in map_f$, and π and τ 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.

6.2 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, [14] 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 comp-equiv*(gcd_1, gcd_2) *in Example 5, it*

is left to check Premise (6.1.3), i.e.,

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

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:

$$\begin{array}{l} (T_{gcd_1} \wedge T_{gcd_2} \wedge (a = x) \wedge (b = y)) \rightarrow \\ (\ ((y_0 > 0) \leftrightarrow (b_0 \neq 0)) \wedge \qquad\qquad\qquad\qquad //\text{Equal guards} \\ \quad ((y_0 > 0) \rightarrow ((b_0 = y_0) \wedge (a_1 = z_0 \mod y_0)))\ ) \quad //\text{Equal inputs} \end{array} \tag{6.2}$$

where T_{gcd_1} and T_{gcd_2} are as defined in Eq. (5.4) and (5.5).

□

6.3 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, π_1 and π_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 π_1 and π_2 are argument-equivalent. Without loss of generality assume also that π_1 is finite and π_2 is not.

Consider the computation π_1. We continue as in the proof of Theorem 1. We construct a computation π_1' in F^{UP}, which is the same as π_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($callee[c_F]$), which receives and returns the same values as received and returned by c_F. In a similar way we construct a computation π_2' in G^{UP} corresponding to π_2. By Premise (6.1.2), any pair of calls to some procedures p_1 and p_2 (related by map_f) satisfy *comp-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 π_1' and π_2' satisfy the congruence condition (see (5.1)). As in the proof of the (PROC-P-EQ) rule, this is sufficient to conclude that π_1' and π_2' are valid subcomputations through F^{UP} and G^{UP} (but not necessarily closed).

By Premise (6.1.3) of the rule and the soundness of the underlying proof system $\mathbb{L}_{\mathbb{UP}}$, π_1' and π_2' satisfy the condition in *reach-equiv*(F^{UP}, G^{UP}) . It is left to show that this implies that π_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 π_1 does not execute any **call** statements. Assume that π_2 executes some call statement c_G in G. Since by Premise (6.1.3) *reach-equiv*(F^{UP}, G^{UP}) holds, then there must be some call c_F in F such that $\langle callee[c_F], callee[c_G]\rangle \in map_f$ and some configuration $C_1 \in \pi_1'$ such that current-label$[C_1] =$ before$[c_F]$ (i.e., π_1' reaches the c_F call). But this is impossible as n_1 is a leaf. Thus π_2 cannot be infinite.

Step:

1. Assume (by the induction hypothesis) that if π_1 is a finite computation with stack-level tree t_1 of depth $d < i$ then any π_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 π_1 with t_1 of depth $d = i$.

2. Let $\widehat{\pi}_2$ be some subcomputation of π_2 from level $d[n_2]+1$, C_2 be the configuration in π_2 which comes immediately before $\widehat{\pi}_2$ ($C_2 = \text{pred}[\widehat{\pi}_2]$). 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 (6.1.3) *reach-equiv*(F^{UP}, G^{UP}), there must be some call c_F in F such that $\langle callee[c_F], callee[c_G]\rangle \in map_f$ 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 π_1 from level $d[n_1]+1$ which starts immediately after C_1 ($C_1 = \text{pred}[\widehat{\pi}_1]$]).

4. $\widehat{\pi}_1$ is finite because π_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 π_2 *from* level $d[n_2]+1$ are finite. By definition, all subcomputations of π_2 *at* level $d[n_2]$ are finite. Therefore π_2 is finite.

□

6.4 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[8].

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

[8]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.

- 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.
- 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:

$$T_{Eval_1} = \left(\begin{array}{ll} a_0 = a & \wedge \\ (isleaf(a_0) \rightarrow r_1 = a_0) & \wedge \\ (\neg\, isleaf(a_0) \wedge isplus(a_0) \rightarrow P(a_0, r_1)) & \wedge \\ (\neg\, isleaf(a_0) \wedge \neg\, isplus(a_0) \wedge ismult(a_0) \rightarrow M(a_0, r_1)) & \wedge \\ r = r_1 & \end{array} \right)$$

$$T_{Eval_2} = \left(\begin{array}{ll} x_0 = x & \wedge \\ (isleaf(x_0) \rightarrow y_1 = x_0) & \wedge \\ (\neg\, isleaf(x_0) \wedge ismult(x_0) \rightarrow M(x_0, y_1)) & \wedge \\ (\neg\, isleaf(x_0) \wedge \neg\, ismult(x_0) \wedge isplus(x_0) \rightarrow P(x_0, y_1)) & \wedge \\ y = y_1 & \end{array} \right)$$

```
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
```

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

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

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

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

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

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

$$\begin{aligned} &\forall l_1, l_2 (doplus(l_1, l_2) = doplus(l_2, l_1) \wedge domult(l_1, l_2) = domult(l_2, l_1)) \\ &\forall t_1 (isleaf(t_1) \rightarrow \neg\, isplus(t_1) \wedge \neg\, ismult(t_1)) \\ &\forall t_1 (isplus(t_1) \rightarrow \neg\, ismult(t_1) \wedge \neg\, isleaf(t_1)) \\ &\forall t_1 (ismult(t_1) \rightarrow \neg\, isleaf(t_1) \wedge \neg\, isplus(t_1)) \end{aligned}$$

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:

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

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:

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

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:

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

7 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.

7.1 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.[9] ◇

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 Section 4.1. 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:

[9] 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.

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

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

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 π, we denote by InSeq$[\pi]$ its input sequence and by OutSeq$[\pi]$ its output sequence. For a finite computation π, we denote by $\Delta\overline{R}[\pi]$ the inputs consumed along π, and by $\Delta\overline{W}[\pi]$ the outputs written during π.

Definition 17 (input-equivalence of subcomputations with respect to procedures)**.** Two subcomputations π_1' and π_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.

7.2 Rule (REACT-EQ)

Figure 11 presents rule (REACT-EQ), which can be used for proving equivalence of reactive procedures.
We prove the soundness of this rule in the next subsection.

$$\frac{\begin{array}{ll}(7.1.1)\ \forall\langle F, G\rangle \in map_f.\ \{ & \\ (7.1.2) \quad \vdash_{\mathbb{L}_{\mathbb{UP}}} & \textit{return-values-equiv}(F^{UP}, G^{UP})\ \wedge \\ (7.1.3) \quad \vdash_{\mathbb{L}_{\mathbb{UP}}} & \textit{input-suffix-equiv}(F^{UP}, G^{UP})\ \wedge \\ (7.1.4) \quad \vdash_{\mathbb{L}_{\mathbb{UP}}} & \textit{call-output-seq-equiv}(F^{UP}, G^{UP})\ \}\end{array}}{(7.1.5) \quad \forall\langle F, G\rangle \in map_f.\ \textit{reactive-equiv}(F,\ G)}\ (\textsc{react-eq}) \tag{7.1}$$

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

7.3 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
π, τ	subcomputations in P_1 and P_2, respectively,
t_1, t_2	stack-level trees of π and τ, 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 π and τ are input equivalent and that $\langle F, G\rangle \in map_f$.

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

then

1) τ *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 π in the premise and τ 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 (7.1.4) holds, τ 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). π and τ 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 π 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 'in-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 5,6,7 in the drawing).

Derive a computation π' in F^{UP} from π as follows. First, set $\text{first}[\pi'] = \text{first}[\pi]$. 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 π each subcomputation $\hat{\pi}_j \in \mathit{from}(\pi, d_1)$ caused by a call c_F, with a subcomputation $\hat{\pi}'_j$ through an uninterpreted procedure UP($\mathit{callee}[c_F]$), which returns the same value as returned by c_F. A small adjustment to $\overline{R}$ and $\overline{W}$ in π' is required since the uninterpreted procedures do not consume inputs or generate outputs. Hence, $\overline{R}$ remains constant in π' 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 τ' in G^{UP} corresponding to τ.

In the course of the proof we will show that π' and τ' 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 π and τ. This will prove the step of Lemma 1.

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

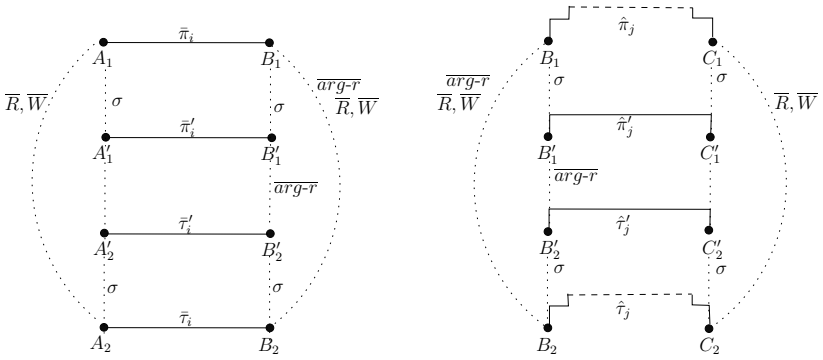

Figure 12: (*left*) 'at-level' subcomputations – a diagram for Lemma 2. (*right*) 'from-level' subcomputations – a diagram for Lemma 3

$$A_1 = \text{first}[\bar{\pi}_i],\ A'_1 = \text{first}[\bar{\pi}'_i],\ A'_2 = \text{first}[\bar{\tau}'_i],\ A_2 = \text{first}[\bar{\tau}_i],$$
$$B_1 = \text{last}[\bar{\pi}_i],\ B'_1 = \text{last}[\bar{\pi}'_i], B'_2 = \text{last}[\bar{\tau}'_i],\ B_2 = \text{last}[\bar{\tau}_i]\ .$$

The figure shows the in-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 in-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 $\bar{\tau}'_i$,

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** $p_1(\overline{e}_1; \overline{x}_1)$ *then* $\bar{\tau}_i$ *ends with* **call** $p_2(\overline{e}_2; \overline{x}_2)$ *and* $\langle p_1, p_2 \rangle \in map_f$
4) *If* $\bar{\pi}'_i$ *ends with a* **call** *statement, then* $B'_1.\sigma[\overline{e}_1] = B'_2.\sigma[\overline{e}_2]$
5) *If* $\bar{\pi}_i$ *ends with a* **call** *statement, then* $B_1.\sigma[\overline{e}_1] = B_2.\sigma[\overline{e}_2]$
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 π, π', τ' and τ, 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 π' to the end of $\bar{\pi}'_i$ and from the beginning of τ' 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** $\mathrm{UP}(p_1)(\overline{e}_1; \overline{x}_1)$. Then, by *call-output-seq-equiv*(F^{UP}, G^{UP}) (premise 7.1.4 of (REACT-EQ)), $\bar{\tau}'_i$ must end with **call** $\mathrm{UP}(p_2)(\overline{e}_2; \overline{x}_2)$ where $\langle p_1, p_2\rangle \in map_f$, 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 7.1.3), 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 computations, 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 π, 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$[B_2] = before[c_G]$,*

$c_F = \textbf{call } p_1(\overline{e}_1; \overline{x}_1)$, *and* $c_G = \textbf{call } p_2(\overline{e}_2; \overline{x}_2)$. *Then*

if

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) $\hat{\pi}_j$ *has a stack-level tree of depth at most* $d-1$
8)$\langle p_1, p_2\rangle \in map_f$,

then

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) $\hat{\tau}_j$ *has a stack-level tree of depth at most* $d-1$.

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 (5.1). As valuations of other variables by $\sigma|_{d_1}$ and $\sigma|_{d_2}$ are unchanged by subcomputations at 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* $A_1 = \mathit{first}[\bar{\pi}_i], A_2 = \mathit{first}[\bar{\tau}_i], A_1' = \mathit{first}[\bar{\pi}_i']$ *and* $A_2' = \mathit{first}[\bar{\tau}_i']$ *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:*

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) $A_1.\overline{W} = A_2.\overline{W}$

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 π and τ are input equivalent (between themselves and with π' and τ'). Consequent 4 stems from Premise 3 of Lemma 1.

Step: Consider in π 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 π', $(\bar{\tau}_i', \hat{\tau}_j')$ in τ', and finally $(\bar{\tau}_i, \hat{\tau}_j)$ in τ.

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 π', τ', π and τ. 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 τ 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 π' and τ'. 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 *return-values-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 π and τ are infinite (recall that by Lemma 1 π is infinite if and only if τ 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 π and τ respectively). Figure 13 presents a possible part of π 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:

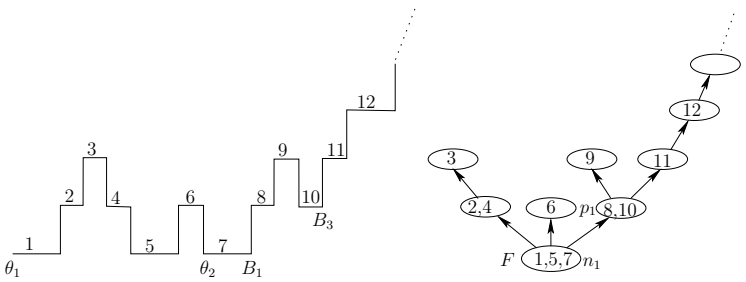

Figure 13: (*left*) A part of an infinite computation π and (*right*) its corresponding stack-level tree t_1. The branch on the right is infinite. The notation correspond to Lemma 5.

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

Lemma 5. *Let π and τ 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$[B_1]$ = before[**call** $p_1(\overline{e}_1;\overline{x}_1)$] and current-label$[B_2]$ = before[**call** $p_2(\overline{e}_2;\overline{x}_2)$]. Then*

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

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 OutSeq$[\pi]$ = 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 θ_1 and θ_2 in the left drawing), and Lemma 2 (for example, between points θ_2 and B_1 in the left drawing) with respect to subcomputations starting at the beginning of F and G and ending at B_1, B_2.

By consequent 3 of Lemma 2 $\langle p_1, p_2 \rangle \in map_f$, 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 at 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). □

7.4 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 $\mathbb{D} \doteq \mathbb{N} \cup \{\text{'+'},\text{'*'},\text{'('},\text{')'}\}$, 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^{L} and sum^{R} receive the value of the sum until now in the formal arguments v^{L} or v^{R} respectively, add to it the value of the next product that they receive in variable r^{L} or b^{R} (from calls to $prod^{\text{L}}$ or $prod^{\text{R}}$), and if the next symbol is ')' they output the sum and return it in variable r^{L} or r^{R} respectively. If the next symbol is '+' they recursively call sum^{L} or sum^{R} to continue the summation.

Similarly, the procedures $prod^{\text{L}}$ and $prod^{\text{R}}$ receive the value of the product up to now in formal argument v^{L} and v^{R}, multiply it by the value of the next natural

number or expression in parentheses (received from num^{L} or num^{R} in variable r^{L} or d^{R}), and get the next symbol in op^{L} or op^{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^{L} or r^{R}).

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

We use the (REACT-EQ) rule to prove reactive equivalence of sum^{L} and sum^{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 $UP(getop^{\mathrm{L}}) = UP(getop^{\mathrm{R}}) = U_g$.

Below 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.

$$\left(\begin{array}{ll} v_0^{\mathrm{L}} = v^{\mathrm{L}} & \wedge \\ U_p(1; r_1^{\mathrm{L}}, op_1^{\mathrm{L}}) & \wedge \\ r_2^{\mathrm{L}} = r_1^{\mathrm{L}} + v_0^{\mathrm{L}} & \wedge \\ (op_1^{\mathrm{L}} = \text{`)'} \ \rightarrow \ out_1 = r_2^{\mathrm{L}}) & \wedge \\ (op_1^{\mathrm{L}} = \text{`+'} \ \rightarrow \ U_s(r_2^{\mathrm{L}}; r_3^{\mathrm{L}})) & \wedge \\ (op_1^{\mathrm{L}} \neq \text{`+'} \ \rightarrow \ r_3^{\mathrm{L}} = r_2^{\mathrm{L}}) & \wedge \\ r^{\mathrm{L}} = r_3^{\mathrm{L}} & \end{array}\right) \qquad \left(\begin{array}{ll} v_0^{\mathrm{R}} = v^{\mathrm{R}} & \wedge \\ U_p(1; b_1^{\mathrm{R}}, op_1^{\mathrm{R}}) & \wedge \\ & \\ (op_1^{\mathrm{R}} = \text{`+'} \ \rightarrow \ U_s(v_0^{\mathrm{R}} + b_1^{\mathrm{R}}; r_1^{\mathrm{R}})) & \wedge \\ (op_1^{\mathrm{R}} \neq \text{`+'} \ \rightarrow \ r_1^{\mathrm{R}} = v_0^{\mathrm{R}} + b_1^{\mathrm{R}}) & \wedge \\ (op_1^{\mathrm{R}} = \text{`)'} \ \rightarrow \ out_2 = v_0^{\mathrm{R}} + b_1^{\mathrm{R}}) & \wedge \\ r^{\mathrm{R}} = r_1^{\mathrm{R}} & \end{array}\right)$$

$\boldsymbol{T_{sum^{\mathrm{L}}}}$ $\qquad$ $\boldsymbol{T_{sum^{\mathrm{R}}}}$

```
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
```

Figure 14: Two reactive calculator programs (labels were removed for better readability). The programs output a value every time they encounter the ')' symbol.

$$\left(\begin{array}{lll} v_0^{\mathrm{L}} = v^{\mathrm{L}} & & \wedge \\ U_n(r_1^{\mathrm{L}}) & & \wedge \\ r_2^{\mathrm{L}} = r_1^{\mathrm{L}} * v_0^{\mathrm{L}} & & \wedge \\ U_g(op_1^{\mathrm{L}}) & & \wedge \\ (op_1^{\mathrm{L}} = \text{`*'} & \rightarrow \; U_p(r_2^{\mathrm{L}}; r_3^{\mathrm{L}}, op_2^{\mathrm{L}})) & \wedge \\ (op_1^{\mathrm{L}} \neq \text{`*'} & \rightarrow \; (r_3^{\mathrm{L}} = r_2^{\mathrm{L}} \wedge & \\ & \quad op_2^{\mathrm{L}} = op_1^{\mathrm{L}})) & \wedge \\ r^{\mathrm{L}} = r_3^{\mathrm{L}} \wedge op^{\mathrm{L}} = op_2^{\mathrm{L}} & & \end{array}\right) \left(\begin{array}{lll} v_0^{\mathrm{R}} = v^{\mathrm{R}} & & \wedge \\ U_n(d_1^{\mathrm{R}}) & & \wedge \\ U_g(op_1^{\mathrm{R}}) & & \wedge \\ (op_1^{\mathrm{R}} = \text{`*'} & \rightarrow \; U_p(v_0^{\mathrm{R}} * d_1^{\mathrm{R}}; & \\ & \quad r_1^{\mathrm{R}}, op_2^{\mathrm{R}})) & \wedge \\ (op_1^{\mathrm{R}} \neq \text{`*'} & \rightarrow \; (r_1^{\mathrm{R}} = v_0^{\mathrm{R}} * d_1^{\mathrm{R}} \wedge & \\ & \quad op_2^{\mathrm{R}} = op_1^{\mathrm{R}})) & \wedge \\ r^{\mathrm{R}} = r_1^{\mathrm{R}} \wedge op^{\mathrm{R}} = op_2^{\mathrm{R}} & & \end{array}\right)$$

$\boldsymbol{T_{prod^{\mathrm{L}}}}$ $\boldsymbol{T_{prod^{\mathrm{R}}}}$

$$\left(\begin{array}{lll} i_0^{\mathrm{L}} = in_1 & & \wedge \\ (i_0^{\mathrm{L}} = \text{`('} & \rightarrow \; U_s(0; i_1^{\mathrm{L}})) & \wedge \\ (i_0^{\mathrm{L}} \neq \text{`('} & \rightarrow \; i_1^{\mathrm{L}} = i_0^{\mathrm{L}}) & \wedge \\ i^{\mathrm{L}} = i_1^{\mathrm{L}} & & \end{array}\right) \left(\begin{array}{lll} i_0^{\mathrm{R}} = in_2 & & \wedge \\ (i_0^{\mathrm{R}} = \text{`('} & \rightarrow \; U_s(0; n_1^{\mathrm{R}})) & \wedge \\ (i_0^{\mathrm{R}} \neq \text{`('} & \rightarrow \; n_1^{\mathrm{R}} = i_0^{\mathrm{R}}) & \wedge \\ n^{\mathrm{R}} = n_1^{\mathrm{R}} & & \end{array}\right)$$

$\boldsymbol{T_{num^{\mathrm{L}}}}$ $\boldsymbol{T_{num^{\mathrm{R}}}}$

$$\left(\begin{array}{ll} op_0^{\mathrm{L}} = in_1 & \wedge \\ op^{\mathrm{L}} = op_0^{\mathrm{L}} & \end{array}\right) \left(\begin{array}{ll} op_0^{\mathrm{R}} = in_2 & \wedge \\ op^{\mathrm{R}} = op_0^{\mathrm{R}} & \end{array}\right)$$

$\boldsymbol{T_{getop^{\mathrm{L}}}}$ $\boldsymbol{T_{getop^{\mathrm{R}}}}$

- **Premise 7.1.2 (*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:

$$\begin{array}{lll} (v^{\mathrm{L}} = v^{\mathrm{R}} \wedge T_{sum^{\mathrm{L}}} \wedge T_{sum^{\mathrm{R}}}) & \rightarrow & r^{\mathrm{L}} = r^{\mathrm{R}} \\ (v^{\mathrm{L}} = v^{\mathrm{R}} \wedge T_{prod^{\mathrm{L}}} \wedge T_{prod^{\mathrm{R}}}) & \rightarrow & r^{\mathrm{L}} = r^{\mathrm{R}} \wedge op^{\mathrm{L}} = op^{\mathrm{R}} \\ (in_1 = R_0 \wedge in_2 = R_0 \wedge T_{num^{\mathrm{L}}} \wedge T_{num^{\mathrm{R}}}) & \rightarrow & i^{\mathrm{L}} = n^{\mathrm{R}} \\ (in_1 = R_0 \wedge in_2 = R_0 \wedge T_{getop^{\mathrm{L}}} \wedge T_{getop^{\mathrm{R}}}) & \rightarrow & op^{\mathrm{L}} = op^{\mathrm{R}} \; . \end{array} \tag{7.2}$$

- **Premise 7.1.3 (*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 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 7.1.4 (*call-output-seq-equiv*).** In procedures sum^{L}, sum^{R}, $prod^{\mathrm{L}}$, $prod^{\mathrm{R}}$, num^{L} and num^{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:

$$\begin{aligned}
&(v^{\mathrm{L}} = v^{\mathrm{R}} \wedge T_{sum^{\mathrm{L}}} \wedge T_{sum^{\mathrm{R}}}) \rightarrow \\
&((1 = 1) \wedge \\
&(op_1^{\mathrm{L}} = \text{`)'} \leftrightarrow op_1^{\mathrm{R}} = \text{`)'}) \wedge (op_1^{\mathrm{L}} = \text{`)'} \rightarrow out_1 = out_2) \wedge \\
&(op_1^{\mathrm{L}} = \text{`+'} \leftrightarrow op_1^{\mathrm{R}} = \text{`+'}) \wedge (op_1^{\mathrm{L}} = \text{`+'} \rightarrow r_2^{\mathrm{L}} = v_0^{\mathrm{R}} + b_1^{\mathrm{R}})) \ .
\end{aligned} \tag{7.3}$$

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.

$$\begin{aligned}
&(v^{\mathrm{L}} = v^{\mathrm{R}} \wedge T_{prod^{\mathrm{L}}} \wedge T_{prod^{\mathrm{R}}}) \rightarrow \\
&((op_1^{\mathrm{L}} = \text{`*'} \leftrightarrow op_1^{\mathrm{R}} = \text{`*'}) \wedge (op_1^{\mathrm{L}} = \text{`*'} \rightarrow r_2^{\mathrm{L}} = v_0^{\mathrm{R}} * d_1^{\mathrm{R}}))
\end{aligned} \tag{7.4}$$

$$(in_1 = R_0 \wedge in_2 = R_0 \wedge T_{num^{\mathrm{L}}} \wedge T_{num^{\mathrm{R}}}) \rightarrow \\ ((i_0^{\mathrm{L}} = \text{`(`} \leftrightarrow i_0^{\mathrm{R}} = \text{`(`}) \wedge (i_0^{\mathrm{L}} = \text{`(`} \rightarrow 0 = 0)) \; . \tag{7.5}$$

Using the uninterpreted procedure relations and commutativity of the '+' and '*' operators, one can prove the validity of (7.2) – (7.5). 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-equiv*(sum^{L}, sum^{R}) holds. Consequently we know that the two programs generate the same output sequence when executed on the same arithmetical expression.

8 What the rules cannot prove

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

1. Consider the following equivalent functions, which compute the sum of numbers from 1 to the input n, which is assumed to be positive.

```
int F(unsigned int n) {
    if (n <= 1) return n;
    return n + F(n-1);
}
```

```
int G(unsigned int n) {
  if (n <= 1) return n;
  return n + (n - 1) + G(n-2);
}
```

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

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

```
int F(unsigned int n) {
    if (n <= 0) return 0;
    return n + F(n-1);
}
```

```
int G(unsigned int n) {
  if (n <= 1) return n;
  return n + G(n - 1);
}
```

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.

```
int F(unsigned int n) {
    int loc;
    if (n <= 0) return 0;
    loc = F(n-1);
    if (loc < 0) return -1;
    return n + loc;
}
```

```
int G(unsigned int n) {
  if (n <= 0) return n;
  return n + G(n - 1);
}
```

In this case the 'if' condition in the first program never holds. Yet since the Uninterpreted Functions return arbitrary, although equal, values, they can return a negative value, which will make this 'if' condition hold and as a result make the two isolated functions return different values.

In the first two examples the compared procedures are not reach-equivalent: in the first example due to the different read arguments with which the recursive call is made, and in the second example due to the different condition under which the recursive calls are made. In the third case it is 'dead-code' that fails the equivalence check. It seems that only with invariants such problems can be solved.

Part II

Regression Verification for C Programs

9 The Regression Verification Tool (RVT)

We have implemented a tool called the *Regression Verification Tool* (RVT). It takes as input two C programs and optionally a third file that contains the equivalence specification as defined by the user – the equivalence specification format will be explained in Sect. 9.2. The two programs correspond to the "*two sides*" of the equivalence verification problem. Our convention is that side 0 is the older version of the program and side 1 is the newer version.

The tool is a preprocessor which decomposes the compared programs, generates verification conditions in the form of nonrecursive C programs with assertions that reflect the user's equivalence specification, and uses an external decision procedure to decide them. Thus, it reduces the equivalence problem to a functional verification problem over a restricted type of program. Currently, the decision procedure is CBMC (see Sect. 1.1), but any other decision procedure which can reason about the validity of assertions in bounded-length C programs can be used. Thus, the tool is independent of the choice of the decision procedure.

Before we present the structure of RVT we first present several assumptions about the programs that we can compare. Then, in Sect. 9.2, we present the notion of checkpoints, which are important for describing how the user defines the equivalence specification.

9.1 Assumptions

The tool we developed works on C programs. The reference standard is ANSI C99, but not all features presented in the standard are covered. The unsupported features are:

- arrays,
- pointers to structures which are converted and passed as "void*" type, and
- many standard library functions, most of which handle strings.

In addition, we assume that

- the programs do not include aliasing between pointers which are arguments of a single function, or which are globals that are used by the same function. This is essential when using the three rules. See Sect. 9.3.5 for an example of how aliasing can invalidate our use of rule (PROC-P-EQ).
- all dynamic structures that are passed to a function (through pointer arguments or globals) are trees, i.e., aliasing within dynamic structures is not allowed either. The reason for this assumption is described in Sect. 9.3.5.

9.2 Check points

Deciding program equivalence based only on the output may be

- Misleading: Output may not well-reflect complex computations.
- Impractical: Sometimes it is computationally too hard to prove the equivalence of the two programs as a whole, but it is easy to verify intermediate results.
- Impossible: In some stages of the development the output is not yet implemented or defined.

For this reason the user may want to compare values which are calculated during the execution of the program. Therefore we present the notion of check-points: check-points are locations (labels) of both programs where some values of some user-specified expressions should be equal between the two compared programs.

The user supplies, in a separate directives file, a list of check-point declarations of the form:

$$(\langle label_1;\ cond_1;\ exp_1\rangle;\ \langle label_2;\ cond_2;\ exp_2\rangle)\ . \tag{9.1}$$

Assume first that $cond_1$ and $cond_2$ are equal to TRUE. Then this checkpoint declaration represents the user's specification that exp_1 in the location specified by $label_1$ in the first program, should always be equal to exp_2 in location $label_2$ in the second

program. In other words, the sequence of values of these two expressions in these locations should be equal in all executions on common inputs. Specifying that the outputs of the program are equal is of course a special case of this mechanism.

Now let us generalize for the case that the conditions are not necessarily equal to TRUE. In the program that RVT generates from the two input programs, there is code that updates an array with the value of exp_i each time it reaches $label_i$ provided that $cond_i$ holds, for $i \in \{1, 2\}$. Then there is an assertion that checks the equivalence of these two arrays. We will describe this mechanism in more detail and precision in Sect. 9.3.3.

Channels Check-points can be associated with *channels*. Then, only check points associated with the same channel are compared to one another. The order of the values between different check points associated with the same channel is important and this sequence of values is required to be the same on both sides. But check points for different channels can interleave and may appear in different order on different sides. *Default channel* is used when simple sequential checks are needed. By default, all check-points are related to the default channel and the order that they are reached in both programs must be the same.

Example 8. *Figure 15 contains code of two functions which includes two check points: the labels "check_point_5" and "check_point_4". The line marked by '*' causes a difference between the two programs. RVT recognizes such a difference. The tool recognizes the labels "check_point_5" and "check_point_4" as check points because they appear in the input "directives" file, which appears in Fig. 16.*

The directives file first declares two channels called "digit_chan" and "power_chan". Then, it declares two check points on each side. Each check point declaration begins with the "CP" keyword. The name of each check point is the label in the source code at which the data for this check is collected. Note that in the example above all condition expressions are 1, i.e., the data is collected every time we reach the specified labels. □

```
int power(int x, unsigned y)
{
  int res ;

  for(res=1;y>0;y--) {
     res = res*x ;
   check_point_5:;
  }
  return res ;
}

int sum_hexas(unsigned long n) {
  int sum=0 ;

  while( n>0 ){
    sum += (n & 0xF);
    n >>= 4;

  check_point_4:;
  }
  return sum ;
}
```

```
int power(int x, unsigned y) {
 int res ;
 unsigned i;

 for(i = 0, res=1;i<y;i++) {
   res *= x ;
  check_point_5:;
 }
 return res ;
}

int sum_hexas(unsigned long n)
{
  int sum=0 ;

  for(;n>0;){
    sum += (n & 0xF);
    n >>= 4;

    if( n == 1234 )  // *
      sum=0;
  check_point_4:;
  }

  return sum ;
}
```

Figure 15: Demonstrating check points

```
CHANNEL digit_chan;
CHANNEL power_chan;

SIDE0_CHECK_POINTS() {
  CP check_point_4 = { digit_chan, 1, sum };
  CP check_point_5 = { power_chan, 1, res };
}

SIDE1_CHECK_POINTS() {
  CP check_point_4 = { digit_chan, 1, sum };
  CP check_point_5 = { power_chan, 1, res };
}
```

Figure 16: The directive file contains the declaration of check points

9.3 Structure and Algorithms

9.3.1 An overview of RVT

The description of RVT relies on the notion of call graphs.

Definition 19. A call-graph of a program is a directed graph $G(V, E)$ such that the vertices correspond to functions and for $v_1, v_2 \in V$, $(v_1, v_2) \in E$ if and only if the function corresponding to v_1 calls the function corresponding to v_2.

Recursive and mutually recursive functions appear as cycles in this graph. The root of this graph corresponds to the "main" function and the leafs, if there are any, to functions that do not call any other function.

As described above, RVT receives two C programs P_1 and P_2 and a directives file with checkpoint declarations. It then follows roughly the following steps (the exact program flow depends on the chosen notion of equivalence and the chosen proof strategy, as we describe later in Sect. 9.3.6).

1. **Converting loops to recursion** All loops in P_1 and P_2 are replaced by recursive functions. This process is described in Appendix C.1.

2. **Pairing** Functions on both sides are paired based on syntactic analysis, which will be described in Sect. 9.3.2.

3. **A call-graph based decomposition** Assume for now that the two programs do not contain mutual recursion, and hence their call graphs are DAGs with possible self loops. The proof of equivalence is now conducted in an iterative manner by following the call graphs of P_1 and P_2 bottom up. In each iteration we attempt to prove the equivalence of two subprograms, that we call the *related subprograms*. This iterative process is required both for computational reasons (decomposition of the proof) and for being able to prove the equivalence of recursive procedures. As for the latter, recall that the rules we described in part I of the thesis allow us to prove the equivalence of recursive and mutually recursive

functions by isolation, i.e., by replacing the recursive calls with uninterpreted functions.

This decomposition algorithm will be described in detail in Sect. 9.3.3.

4. **Syntactic check** RVT checks whether the two related subprograms are equivalent syntactically, namely the expression trees of the code on both sides are isomorphic up to renaming of the internal variables (Section 9.3.2 describes this in more detail). If yes then the related subprograms are declared equivalent.

5. **Semantic check** Otherwise, from a pair of related subprograms, RVT constructs a small C program which we call a *check-block* and sends it to the decision procedure (CBMC). The conversion of related subprograms into check-blocks is done in several steps, most of which will be described in more detail in the next subsections.

   (a) Global identifiers (of global functions, variables and types) are renamed. This enables us to put the two related subprograms in the same check-block without name-space collisions.

   (b) Code is inserted at the check-point labels that collect the *exp* values specified in the check point during the execution of the check-block. The code for each check-point is guarded by the condition expression associated with the check-point.

   (c) When proving rule (M-TERM) or (REACT-EQ), additional code is inserted in order to collect the data necessary for proving call-sequence equivalence and reach-equivalence (see definitions 14 and 30). We discuss these issues in Sects. 9.4.2 and 9.4.3.

   (d) Code is inserted to declare and initialize structures and arrays that are used to store and compare the collected values. A separate array is specified for each channel.

(e) Since uninterpreted functions are not supported directly by CBMC, we add to the check-block code that implements them – essentially we replace the bodies of functions that we wish to make uninterpreted with code that enforces the congruence relation. This is described in more detail in Sect. 9.4.1.

(f) If uninterpreted functions are used in the compared code, and the functions that they replaced include check points, we add code that guarantees that the order of reaching the check points and the call to uninterpreted functions is equivalent on both sides. This is explained in Sect. 9.4.2.

(g) Finally, a "main" function is added which contains the code to initialize the inputs to non-deterministic but equal values, executes the related subprograms one after the other and asserts that the output values and the expressions written to the channels are equal.

The check-block is sent to the decision procedure (CBMC). If it decides that no path in the check-block can violate one of the assertions, the root functions of the related subprograms are declared partially equivalent. They can then be replaced by uninterpreted functions when RVT attempts to prove the equivalence of their callers. If, however, the proof of equivalence fails, then when checking their callers, RVT leaves the call to these functions as is and adds their code to the related subprograms. We call this operation *logical inlining*[10], since it enables us to use rule (PROC-P-EQ) as if the whole related subprogram is a single function. This operation is only done if the callees are nonrecursive.

6. **Fallback solution: k-equivalence** If both syntactic and semantic checks fail with a given pair of related subprograms (even after inlining, or when inlining is impossible), RVT attempts to prove the k-equivalence of the corresponding functions. During the k-equivalence check functions which were previously

[10]Inlining is a known term referring to embedding of the code of a function into its caller. We do not do this embedding physically, rather only add the code to the related subprograms, and hence we call it 'logical' inlining.

proved to be equivalent can still be replaced by uninterpreted functions. Other descendants can be added as is because the programs should not necessarily be nonrecursive. k-equivalence, similarly to semantic checks, is decided with CBMC.

Some more sophisticated proof strategies, which employ other combinations of semantic, syntactic and k-equivalence are also supported by RVT. These are described in Sect. 9.3.6.

When any of the semantic or k-equivalence checks fails, CBMC generates a counterexample. This counterexample consists of two runs through the checked related subprograms which begin with the same input values but end with different values at output or check-points, and hence fail an assertion in the check-block. The counterexample is concrete (actual executions of the compared programs) only when a k-equivalence check is performed without uninterpreted functions.

To summarize, the verification conditions generated by RVT are simply small C programs without loops or function calls, which contain portions of code from the two compared programs, code that computes and stores the values in the check points, and assertions that declare their pairwise equivalence. These assertions are then checked with CBMC. Hence, the equivalence problem is reduced to an iterative application of functional verification over a restricted type of program.

9.3.2 Pairing functions and variables between sides

RVT builds map_f using a mechanism that pairs functions. A similar pairing is done between global variables. In practice it is not always able to achieve bijective pairing, and as a result some of the functions and global variables are not paired.

Pairing functions is needed for two reasons. First, the decomposition algorithm works in the granularity of functions; Second, functions are isolated from their callees (by replacing paired functions with equal uninterpreted functions) in the process of proving the equivalence of recursive functions, when using one of the three rules that

were described in the first part of the thesis (recall that all rules were defined with respect to a full mapping between the functions).

Pairing global variables between the two sides is necessary due to a related reason. Whenever we use uninterpreted functions, we need to enforce the congruence relation, which means that we need to recognize the variables that are the inputs and outputs of the functions. These include not only the actual arguments of the functions but also the global variables which are accessed by the function code or by any of the descendants of this function in the call graph. Therefore we need to pair the global variables between the two sides.

Pairing is done recursively, in a manner reminiscent of computing congruence closure. The algorithm works on the parse trees of both programs, and, when it decides to pair two nodes in these trees (where a node can be either a variable, a function, or a type), it adds a pointer in both directions. Note that wrong pairing does not affect soundness: pairing is used for generating the verification conditions, and hence wrong pairing can only result in failing to prove program equivalence.

The pairing algorithm works top-down: it initially attempts to pair global variables and functions according to their names and types. Then, within paired functions that are also syntactically equivalent up to variable names, it attempts to pair elements that appear in isomorphic locations. If these elements were already paired we just check that the pairing according to this function agrees with the previous one. Otherwise we issue a warning. This process is repeated until no new pairing is discovered.

A pair of nodes n_1,n_2 are paired in our pairing algorithm if one of the following conditions hold:

1. n_1, n_2 are the same generic C types.

2. n_1, n_2 are types defined by typedef with the same name and their types are paired.

3. n_1, n_2 are complex types with the same name where the expression trees that

define them are isomorphic, and isomorphic nodes representing identifiers are paired.

4. n_1, n_2 are global variables with the same name and their types are paired.

5. n_1, n_2 are functions that have

   - the same name,
   - the same list of types of formal arguments,
   - the same return type,
   - there is a bijective relation $\mathcal{B}$ between the respective sets of global variables read (written) by the two functions, such that $v_1, v_2 \in \mathcal{B}$ if and only if v_1, v_2 are paired.

6. $n_1, \ldots, n_i$ and $n'_1, \ldots, n'_i$ are paired (pair-wise) if they are the arguments of paired functions. Note that the decision to pair functions is based of the *types* of these arguments, i.e., the arguments are related together with their function.

7. n_1 and n_2 are variables assigned or used in isomorphic expression trees, whose other nodes on both sides are already paired to one another.

8. n_1 and n_2 are complex types (not necessarily with the same name) with isomorphic expression trees, and they were used to qualify some paired variables or functions.

9. n_1 and n_2 are struct components that have the same index (in the component order of paired struct types) and the same type.

10. n_1 and n_2 are objects pointed to by paired pointers.

11. n_1 and n_2 are array items with the same index and their base pointers are paired.

12. n_1 and n_2 are struct objects generated by malloc and their address was passed after creation to paired pointers.

We apply these rules iteratively until no additional relation is added to the pairing. Note that these heuristics depend on the order in which we apply the rules and traverse the code. For example, consider a situation where on side 0 two different global variables $v1$ and $v2$ were used in different unrelated functions $f1$ and $f2$ respectively. Suppose that on side 1 there is a single global variable $v3$ in the role of both these variables. If $f1$ is compared between the sides before $f2$ then $v3$ will be related to $v1$. Otherwise $v3$ will be related to $v2$. Note that there is no valid pairing between the variables in this case.

In addition to the pairing heuristic described above, RVT allows the user to declare parts of the pairing manually in a separate file. It also allows the user to force conversion of functions to uninterpreted functions in all check-blocks in which they are used.

9.3.3 The main algorithm: simple recursion

The equivalence check in RVT, in its basic form, is presented in Algorithm 1. It is based on traversing bottom-up the call graphs of the two programs to be compared. This algorithm can be applied directly to two call graphs without loops of length larger than 1 (i.e., no mutual recursion). The more general case is considered in Sect. 9.3.4.

The algorithm as presented refers to a simplified problem, as follows:

- The algorithm is targeted for proving partial equivalence based on rule (PROC-P-EQ). Rules (M-TERM) and (REACT-EQ) will be discussed in Sects. 9.4.2 and 9.4.3.

- The algorithm does not handle check points: it merely attempts to prove that the two programs return equal values given the same inputs. The treatment of checkpoints is discussed at the end of this section.

- As indicated in Sect. 9.3.1, when RVT fails to prove equivalence based on rule (PROC-P-EQ), it uses CBMC to check for k-equivalence. In some cases CBMC is even able to prove *total equivalence* (for this we need to add assertions that state that no execution requires more than k iterations, and prove them). For simplicity we ignore this mode here. Various strategies of combining the algorithm with k-equivalence are described in Sect. 9.3.6.

We assume that the two programs are given to the algorithm after all loops were replaced with recursive functions (the translation procedure is described in Appendix C.1). In addition, Algorithm 1 expects to receive a set *map* of pairs of functions that need to be compared (normally $map = map_f$). This set does not necessarily include all functions, but it should include as minimum all *recursive* functions.

Initially all nodes are unmarked. Each element $\langle f, g\rangle \in map$ is then annotated with either "Equivalent" or "Failed" corresponding to the case that equivalence between f and g has been proven, or that the proof failed, respectively. If the algorithm aborts before reaching all pairs the partial marking is still valid.

The algorithm progresses bottom-up on the call graphs, and updates the labels to "Equivalent" or "Failed". The progress on the graph is made by a procedure NEXT_UNMARKED_PAIR() (not presented) which returns the next unmarked pair in *map*, according to a BFS order on the reversed call graph of side 0. This procedure aborts if either all pairs are already marked, or it finds that the call-graphs are inconsistent (inconsistency means that there are two pairs of functions $\langle f, f'\rangle \in map$ and $\langle g, g'\rangle \in map$ such that f is a descendant of g but f' is an *ancestor* of g').[11]

We continue describing the labeling procedure. A syntactically equivalent pair such that all its children are already marked by "Equivalent" (or, as a special case, with no children in the respective call graphs), is marked "Equivalent". Otherwise, in line 5, the pair is checked for equivalence semantically.

The semantic equivalence check is conducted by verifying, with CBMC, that various assertions hold in a single loop-free and recursion-free C program *check-block(f, g)*

[11] We consider inconsistency to be an unrealistic scenario in practice.

that RVT constructs. CBMC returns TRUE if the assertions hold and FALSE otherwise.

We proceed by describing check-block(f, g). Consider the maximal connected sub-DAG rooted at f that contains only functions that are unpaired or marked "Failed". Let S_f denote this set of functions (excluding f). S_g is defined similarly with respect to g. The program check-block(f, g) consists of the following elements:

1. The functions f,g and all functions in S_f, S_g, such that
   - Name collisions in global identifiers of the two programs are solved by renaming.
   - All calls to f, g are replaced with calls to UF(f),UF(g), respectively.
   - For all $\langle h_1, h_2 \rangle \in map$ such that $h_1, h_2 \notin \{S_f \cup S_g\}$, calls to h_1, h_2 are replaced with calls to $UF(h_1), UF(h_2)$. (Observe that $\langle h_1, h_2 \rangle$ is marked "Equivalent").
2. A `main()` function:
   - Assignment of nondeterministic but equal values to inputs of f and g.
   - Calls to f, g.
   - Assertion that the outputs of f and g are equal.

Several notes on the definition of check-block(f, g)

- The check-block is nonrecursive. This is because when a recursive pair $\langle f, g \rangle \in map$ is labeled "Failed" the algorithm aborts in line 8, and hence the code of f, g will not be part of future check-blocks.[12]
- The code of each nonrecursive pair $\langle f, g \rangle \in map$ that is labeled "Failed" is logically inlined when checking the equivalence of their parents, and possibly more ancestors, until reaching a provably equivalent pair or reaching the roots.

[12] In practice RVT switches to k-equivalence in this case rather than aborting.

This enables RVT to prove equivalence in case, for example, that some code was moved from the parent to the child, but together they still perform the original computation.

- The code of a pair $\langle f, g\rangle \in map$ that is proven to be equivalent does not participate in any subsequent check-block. It is replaced with uninterpreted functions in all subsequent semantic checks, or disappears altogether if some ancestor pair is marked "Equivalent" as well in each of its paths to the roots of the related subprograms.

- The replacement of recursive calls of paired functions with uninterpreted functions corresponds to isolation (see Definition 11). Recall that proving equivalence of all paired isolated procedures also proves the partial equivalence of all of them by rule (PROC-P-EQ).

Algorithm 1 A basic call-graph based algorithm for attempting to prove the equivalence of pairs of functions.

Procedure COMPARE()
input: Call graphs CG_1 and CG_2 and a mapping *map*.
output: Marking of pairs in *map* with "Equivalent" or "Failed".

```
1: ⟨f, g⟩ = NEXT_UNMARKED_PAIR()                ▷ Bottom-up. Aborts if none.
2: if ⟨f, g⟩ are syntactically equivalent and all their children are marked by "Equivalent" then
3:     Mark ⟨f, g⟩ by "Equivalent".
4: else
5:     if CBMC(check-block(f, g)) then            ▷ Semantic check
6:         Mark ⟨f, g⟩ "Equivalent".
7:     else
8:         if f, g are recursive then abort.
9:         Mark ⟨f, g⟩ "Failed".
10: goto line 1.
```

Complexity Each pair in *map* is labeled at most once by either "Failed" or "Equivalent". Thus, if $n = |map| = |map_f|$, which, in turn, cannot be larger than the number

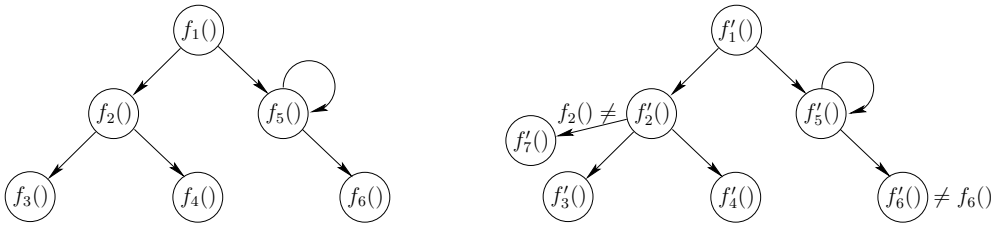

Figure 17: Two call graphs for Example 9

of functions, then the algorithm performs not more than n syntactic and n semantic checks.

Example 9. *Consider the call graphs that are presented in Fig. 17. Assume that for* $i = 1, \ldots, 6$ *we have* $\langle f_i, f'_i \rangle \in map$*. All paired functions are syntactically equivalent except functions* $f_2 \neq f'_2$ *and* $f_6 \neq f'_6$ *as marked on the right side of the figure. We describe step by step the execution of Algorithm 1:*

1. *In line 3 pairs* $\langle f_3, f'_3 \rangle, \langle f_4, f'_4 \rangle$ *are marked "Equivalent".*

2. *The program check-block*(f_2, f'_2) *is sent to CBMC. Now* $S_{f'_2} = \{f'_7\}$ *and hence this program contains also the code of* f'_7*, whereas* f_3, f'_3, f_4, f'_4 *are replaced by uninterpreted functions. Assume that this semantic check fails. Then the pair* $\langle f_2, f'_2 \rangle$ *is marked "Failed".*

3. *The program check-block*(f_6, f'_6) *is sent to CBMC. Assume that the check fails and hence the pair is marked "Failed".*

4. *The program check-block*(f_5, f'_5) *is sent to CBMC. Since* $S_{f_5} = \{f_6\}$ *and* $S_{f'_5} = \{f'_6\}$*, this program contains also* f_6, f'_6*. The recursive calls are replaced with uninterpreted functions. Assume that this time the check succeeds. Then* $\langle f_5, f'_5 \rangle$ *is marked "Equivalent" in line 6.*

5. *The program check-block*(f_1, f'_1) *is sent to CBMC. Now* $S_{f_1} = \{f_2\}$ *and* $S_{f'_1} = \{f'_2, f'_7\}$*. Hence, the respective call subgraphs contain also* f_2, f'_2 *and* f'_7*, whereas*

$f_3, f_3', f_4, f_4', f_5, f_5'$ are replaced with uninterpreted functions. Assume this check succeeds. The algorithm marks $\langle f_1, f_1' \rangle$ by "Equivalent".

At this stage all function pairs are marked by either "Equivalent" or "Failed" and the algorithm terminates.

Theorem 4 (Correctness of Algorithm 1)**.** *Pairs marked by "Equivalent" by Algorithm 1 are partially equivalent.*

Proof. We give a proof sketch.
For each $\langle f, g \rangle$ which the algorithm marks "Equivalent":

1. Let f_E be the function f with all the functions in S_f logically inlined into f. Define g_E similarly with respect to S_g and g.

2. If $\langle f, g \rangle$ was marked "Equivalent" in line 3, then $f_E = f, g_E = g$ and they are syntactically equivalent. Therefore, $\langle {f_E}^{UP}, {g_E}^{UP} \rangle$ (i.e., the isolated versions of f and g, respectively) are computationally equivalent.

3. Now consider the case $\langle f, g \rangle$ was marked "Equivalent" in line 5. All pairs in $\langle {f_E}^{UP}, {g_E}^{UP} \rangle$ were successfully proved computationally equivalent by CBMC.

4. Let D_f be the subDAG which contains all successors of f. Define D_g similarly with respect to g.

5. By line 1 when $\langle f, g \rangle$ is marked all the functions in D_f, D_g were already marked before.

6. This means that all isolated pairs in D_f, D_g which were marked "Equivalent" were proved to be computationally equivalent.

7. If we consider D_f, D_g as two compared programs, then by rule (PROC-P-EQ) all such function pairs in D_f, D_g are partially equivalent.

8. Since $\langle f_E, g_E \rangle$ are partially equivalent then so is $\langle f, g \rangle$. This is because inlining does not change the fact that f and g are computationally equivalent.

9. Hence, pairs which are marked "Equivalent" are indeed equivalent.

□

Accounting for check points. If any of the functions in $\{f, g\} \cup S_f \cup S_g$ contain check-points then check-point and channel code are added to check-block(f, g) as was explained in Sect. 9.2. Assertions that the sequences of values written to the channels are equal are added to the `main()` function after calling the two root functions f and g. This is not enough, however, if functions that are replaced with uninterpreted functions include check points, because now these check point values are not part of the sequence. We should verify, then, that the sequence of values and calls to the uninterpreted functions is the same. This is similar to the notion of call-output-seq-equiv that we saw in Sect. 7.1. In Sect. 9.4.2 we describe in more detail how this check is implemented.

9.3.4 The main algorithm: mutually recursive functions

The code above describes the algorithm when there are no mutually recursive functions, i.e., when the call-graphs do not contain any cycles of length > 1. In the presence of mutually recursive functions the call-graphs contain larger cycles, or, more generally, maximal strongly connected components (MSCCs). In an MSCC each function has at least one callee in the same MSCC whose equivalence was not proved yet. But as rule (PROC-P-EQ) suggests we can prove the whole MSCCs equivalent by checking each of its functions in isolation, assuming they are all paired.

In practice not all functions in the MSCCs are paired, and even if they are, requiring all functions in the MSCCs to be equivalent is possibly too strong of a requirement. As we show it is sufficient to prove equivalent a subset of the paired functions that constitute a *feedback vertex set* (a set of nodes that intersect all cycles) in the MSCCs. After proving such a set equivalent, they can be replaced with uninterpreted functions and then the other functions can be checked with the method of the previous section.

Algorithm 2 begins by constructing an *MSCC DAG* for each side:

Definition 20 (MSCC DAG)**.** An MSCC DAG is derived from the call graph by collapsing each MSCC to a single node.

In this graph there is an edge from MSCC m_1 to MSCC m_2 if and only if a function in m_1 calls a function in m_2. There are no self-loops.

Next, in line 2, we try to create a map map_m of all nodes in the MSCC DAGs that represent recursive or mutually recursive functions. If f appears in an MSCC and g in an MSCC on the other side and $\langle f, g\rangle \in map_f$, then we pair these MSCCs. If there is a contradiction in this process or there is no 1-1 and onto mapping between the MSCCs (excluding nonrecursive functions), then RVT reverts to k-equivalence. Otherwise, it progresses bottom up on the MSCCs DAG.

Algorithm 2 Call-graph algorithm with MSCCs.

Procedure CompareWithSCCs()
input: Call graphs CG_1 and CG_2.
output: Marking of pairs in map_f with "Equivalent" or "Failed".

1: Generate MSCC DAGs MD_1 and MD_2 from CG_1 and CG_2.
2: Generate a map map_m between recursive or mutually recursive functions in MD_1 and MD_2 that is consistent with map_f.
3: **if** such map_m does not exist **then** abort.
4: **while** $\exists\ \langle m_1, m_2\rangle \in map_m$ unmarked with all children marked "Covered" **do**
5: Select nondeterministically a subset S of pairs of functions from m_1, m_2 that constitute feedback vertex sets of m_1, m_2.
6: **for all** $\langle f, g\rangle \in S$ **do**
7: **if** not CBMC(check-block'(f, g)) **then** abort.
8: **for all** $\langle f, g\rangle \in S$ **do** mark $\langle f, g\rangle$ as "Equivalent".
9: Let map be all the pairs in m_1, m_2.
10: Remove all outgoing edges from nodes marked "Equivalent" in CG_1, CG_2.
11: Call Compare(CG_1, CG_2, map).
12: Mark $\langle m_1, m_2\rangle$ as "Covered".

In line 5 the algorithm chooses nondeterministically a set S of paired functions that constitute a feedback vertex set of both m_1 and m_2. The nondeterminism is used only for simplifying the description of the algorithm. It can be determinized

by attempting all minimal feedback vertex sets, until one is found that we can prove equivalent.[13] In practice, RVT initially attempts to take S as the entire set of pairs in m_1 and m_2. If this fails, it removes from S pairs that it failed to prove equivalent. This process continues until it succeeds or until S does not intersect all cycles. The success of this step depends on the order of the checks.

Given S, we continue in line 7 by trying to prove the equivalence of each pair in S. The program check-block'(f, g) is very similar to check-block(f, g) that was described in the previous section, the only difference being that more functions are included, as follows.

Now consider the maximal connected subDAG rooted at f that contains only functions that are not marked "Equivalent" and are not in S. Let S'_f denote this set of functions (excluding f). S'_g is defined similarly with respect to g.[14] The construction of the program check-block'(f, g) is identical to that of check-block(f, g), other than the fact that we use S'_f, S'_g rather than S_f, S_g, respectively.

If the proof succeeds for all pairs in S, we call COMPARE in line 11 to attempt to prove equivalence of other functions in the MSCCs. Since COMPARE only works on call graphs without loops larger than 1, we break all cycles in the call graphs in line 10 by removing outgoing edges from nodes that are marked "Equivalent". First, we know that this breaks all cycles in the call graphs because we can reach this line only after marking "Equivalent" all functions in S, which, recall, is a feedback vertex set. Second, recall that these functions are replaced with uninterpreted functions in COMPARE and hence their outgoing edges have no effect on the algorithm.

For simplicity of the presentation of Algorithm 2, we ignored the possibility of proving equivalence by syntactic checks. In practice RVT also performs syntactic checks: if a pair $\langle f, g\rangle \in S$ is syntactically equivalent, and all the functions in S'_f, S'_g are syntactically equivalent pair-wise, then it marks f, g as "Equivalent" and avoids

[13] Although there can be an exponential number of them in the size of the MSCC, observe that large MSCCs in real programs are rare.

[14] The difference from the definition of S_f,S_g that we used in Sect. 9.3.3 is that S'_f, S'_g may include *unmarked* nodes (in m_1, m_2).

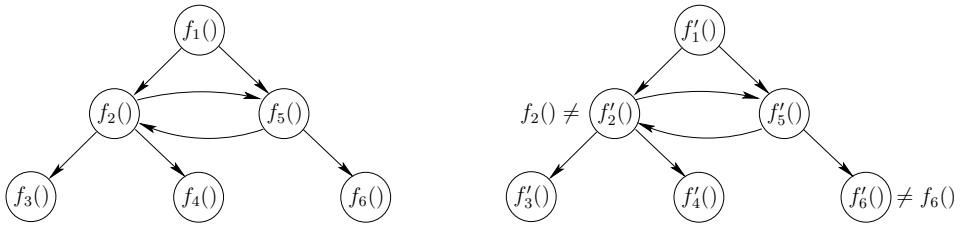

Figure 18: Two call graphs for Example 10

the semantic check.

Example 10. *Consider the call graphs that are presented in Fig. 18. Assume that for* $i = 1, \ldots, 6$ *we have* $\langle f_i, f'_i \rangle \in map_f$*. All paired functions are syntactically equivalent except functions* $f_2 \neq f'_2$ *and* $f_6 \neq f'_6$ *as marked on the right side of the figure. We describe step by step the execution of Algorithm 2:*

1. *The vertices of the MSCC DAGs are (listed bottom-up, left-to-right):* $MD_1 = \{\{f_3\}, \{f_4\}, \{f_6\}, \{f_2, f_5\}, \{f_1\}\}$ *and* $MD_2 = \{\{f'_3\}, \{f'_4\}, \{f'_6\}, \{f'_2, f'_5\}, \{f'_1\}\}$*. The MSCC mapping* map_m *is naturally derived from* map_f*.*

2. *In line 5,* S *can be set to* $\emptyset$ *for both MSCC pairs* $\{f_3\}, \{f'_3\}$ *and* $\{f_4\}, \{f'_4\}$*. Then, in line 11 pairs* $\langle f_3, f'_3 \rangle, \langle f_4, f'_4 \rangle$ *are marked "Equivalent" and their respective MSCCs are marked "Covered".*

3. *Similarly, for MSCCs* $\{f_6\}, \{f'_6\}$*,* S *can be set to* $\emptyset$ *and the pair* $\langle f_6, f'_6 \rangle$ *is checked by* Compare*. Assume that the check fails and hence the pair is marked "Failed". The MSCCs are marked "Covered".*

4. *Assume that for the MSCCs* $\{f_2, f_5\}, \{f'_2, f'_5\}$ *the algorithm chooses* $S = \{f_2, f'_2\}$*.*

5. *The program check-block'*(f_2, f'_2) *is sent to CBMC. In this check* f_3, f'_3*,* f_4, f'_4 *are replaced by uninterpreted functions and the functions* f_5, f'_5*,* f_6, f'_6 *are inlined (*$S'_{f_2} = \{f_5, f_6\}$*,* $S'_{f'_2} = \{f'_5, f'_6\}$*). The calls to* f_2, f'_2 *in* f_5, f'_5 *are replaced with*

uninterpreted functions. Assume that this semantic check succeeds. Then the pair $\langle f_2, f_2' \rangle$ *is marked "Equivalent" in line 8.*

6. *The program check-block'*(f_5, f_5') *is checked by* COMPARE *in line 11, where* $map = \langle f_5, f_5' \rangle$*. In this case* $S'_{f_5} = \{f_6\}$ *and* $S'_{f_5'} = \{f_6'\}$*. Hence this program contains also* f_6, f_6'*, and the calls to* f_2, f_2' *are replaced with uninterpreted functions. Assume that the check succeeds. Then,* $\langle f_5, f_5' \rangle$ *is marked "Equivalent". The MSCCs* $\{f_2, f_5\}, \{f_2', f_5'\}$ *are marked "Covered".*

7. *The pair* $\langle f_1, f_1' \rangle$ *is checked by* COMPARE *(again* $S = \emptyset$*) and marked "Equivalent". Their MSCCs are marked "Covered".*

At this stage all MSCCs are marked "Covered" and the algorithm terminates.

Theorem 5. *Pairs marked by "Equivalent" by Algorithm 2 are partially equivalent.*

Proof. We give a proof sketch.

- Consider some pair of MSCCs m_1, m_2 and a subset S as defined in line 5.
- For each $\langle f, g \rangle \in S$ let f_E be the function f with all the functions in S'_f logically inlined into f. Define g_E similarly with respect to S'_g and g.
- If all pairs in $\langle {f_E}^{UP}, {g_E}^{UP} \rangle$ (i.e., the isolated versions of f_E and g_E, respectively) are successfully proved equivalent in line 7, by rule (PROC-P-EQ) this means that all pairs $\langle f_E, g_E \rangle$ are partially equivalent.
- Since $\langle f_E, g_E \rangle$ are partially equivalent then so is $\langle f, g \rangle$. This is because inlining does not change the fact that f and g are computationally equivalent.
- Hence, pairs in S are marked "Equivalent" only if they are indeed equivalent.
- The only remaining unmarked functions that are reachable from m_1, m_2 are in $(S'_f \cap m_1)$ and $(S'_g \cap m_2)$ (this is because function pairs in S'_f, S'_g outside of m_1, m_2 are already marked "Failed"). By correctness of COMPARE, these pairs

are marked correctly. Note that the call graphs that we send to COMPARE are forests due to line 10, and hence the check-blocks that it checks are nonrecursive.

□

9.3.5 Unbounded structures

Deciding the truth of formulas with uninterpreted functions requires comparing arguments of instances of such functions, or their outputs. If some of these arguments are pointers, such a comparison is meaningless. In this section we describe RVT's method of treating pointer arguments of functions and dynamic data structures, and also show why this method fails in the presence of aliasing.

Whereas in nonpointer variables the comparison is between values, in the case of pointer variables the comparison should be between the data structures that they point to. A dynamic data structure can be represented as a graph whose vertices are structs and edges are the pointers that point from one struct to the other. We call such graph a *pointer-element graph*. Based on this representation we define equality of structures:

Definition 21 (Iso-equal structures)**.** We say that two structures are *iso-equal* if their pointer-element graphs are isomorphic and the values at structs related by the isomorphism are equal.

RVT checks for bounded iso-equality. It makes a simplifying assumption that the data structure is a tree (we were not able to generalize the solution to an arbitrary structure)[15]. Let p_1, p_2 be paired pointer variables that are arguments to the functions

[15]We tried to implement more elaborate solutions, but CBMC was unable to cope with the sizes of the resulting verification goals even for the simplest structures. We tried two alternatives for uninterpreted bounded structure construction: (1) Generate the structure on-the-fly during the run of the function. We used tokens to track which pointers in the function must point to the same object. This info is needed at each dereference operation to determine the value accessed by the dereference. (2) Analyze the function in advance to find all possible structure forms. The Analysis should be done on the SSA (see Sect. 5) form of the function.

As no function contains loops, the check-block code for both the above alternatives should generate only small bounded structures. In both cases we should take into account all possible aliasing inside

that we wish to compare. RVT generates a nondeterministic tree-like data structure of a depth corresponding to a bound (see below) and make both p_1 and p_2 point to it. This guarantees that the input structure is arbitrary but equivalent, up to a bound and under the assumption that it is a tree, on both sides. A similar strategy is activated when we compare p_1 and p_2 that point to an output of the compared functions.

What should be the bound on the depth of the arbitrary data structure? Recall that the code of the related subprograms that we check does not contain loops or recursion, and hence there is a bound on the maximal depth of the items this code can access in any dynamic data structure that is passed to the roots of the related subprograms. It is possible, then, to compute this bound, or at least overestimate it, by syntactic analysis. For example, searching for code that progresses on the structure such as `n = n -> next` for a pointer `n`. Such a mechanism is not implemented yet in RVT, however, and it relies instead on a user-defined bound. When checking k-equivalence, dynamic data structures are unwound k times this bound.

Aliasing. Recall that in general aliasing between pointers that are function arguments is unacceptable by our method. The programs in figures 19 and 20 demonstrate how argument passing by reference with aliasing invalidates the (PROC-P-EQ) rule.

```
void div1(int x, int y,
          int* out_d,
          int* out_r)
{
    if( y == 0 )
      return;
    *out_d = x/y;
    *out_r = x % y;
}
```

```
void div2(int x, int y,
          int* out_d,
          int* out_r)
{
    if( y == 0 )
      return;
    *out_r = x % y;
    *out_d = x/y;
}
```

Figure 19: Two allegedly equivalent C functions.

If we perform a semantic check on the two functions in figure 19 we will be convinced that "out_r" and "out_d" point to equal output values on both sides. This

the structure and generate nondeterministic code which can generate each of them. This was the part of the solutions which CBMC failed to support.

```
int main() {
    int result;
    div1(5, 2,
         &result, &result);
    return result;
}
```

```
int main() {
    int result;
    div2(5, 2,
         &result, &result);
    return result;
}
```

Figure 20: These two programs call the `div1` and `div2` procedures from Fig. 19, but return different results due to aliasing.

is because when the addresses "out_r" and "out_d" of the output values are different, the only difference between the functions is the order of calculation. But when we use the functions as in figure 20 both these variables will point to the same "result" variable and its value at the end of `main()` will be $x\%y$ on one side but x/y on the other.

It is possible to add an assertion to the generated check-block that checks that no aliasing is possible between function arguments, and this way guarantee soundness.[16]

9.3.6 Proof strategies

RVT supports several proof strategies which try various combinations of semantic-equivalence and k-equivalence checks. These are:

k-equivalence The only check-block contains all functions in both programs, and CBMC unwinds all recursions k times. A counterexample is real as long as external (bodyless) functions are implemented or have no influence. CBMC has an option of adding assertions (called "unwinding assertions") with which we can check that the given bound k is sufficient, i.e., no execution requires more than k iterations. The assertions state that the guard of a loop construct is not satisfied at the end of the k-th iteration. When using this option, a 'verification successful' result implies that the programs are totally equivalent.

Alternating A hybrid strategy between pure k-equivalence as described above and

[16] RVT does not support this option yet.

pure semantic checks. Whenever the semantic check fails or is impossible because the related subprograms are recursive, we invoke k-equivalence on these related subprograms. There can be three outcomes to this check:

Equivalent CBMC is able to prove k-correctness while using unwinding assertions as described above,

k-equivalent The same as above, without the unwinding assertions, and

Failed k-equivalence CBMC finds a counterexample to k-equivalence. Note that this does not necessarily mean that the two functions are indeed k-different, because the counterexample can be spurious due to the use of uninterpreted functions in the check-block (recall that uninterpreted functions abstract the real computations).

In the first case RVT marks the pair of nodes "Equivalent". In the second it marks them "k-Equivalent", and continues as if they were equivalent (replaces them with uninterpreted functions when checking their parents), but from hereon, all its ancestors can only be proven to be k-equivalent. In the last case the pair is marked "Failed".

Note that since semantic checks (as described in Sect. 9.3.3) amounts to checking a check-block with no recursion, it corresponds to a k-equivalence check with bound equal to 1. Hence RVT in practice invokes CBMC with this file and a bound k, which becomes relevant only if the check-block is recursive. Hence for each related subprograms pair only a single proof effort is made.

Hybrid RVT first activates syntactic and semantic checks as described in Algorithm 2. If it is able to prove the equivalence of `main()` then the two programs are partially equivalent. Otherwise it switches to k-equivalence, while still replacing functions that were proved equivalent in the first phase with uninterpreted functions. Here too, a counterexample can be spurious.

9.4 uninterpreted functions, call-sequence equivalence and reach-equivalence

In this section we consider the implementation of uninterpreted functions, call-sequence equivalence and reach-equivalence, in this order.

9.4.1 Implementation of uninterpreted functions

CBMC does not support uninterpreted functions directly, as its input language is C. To implement uninterpreted function we replace the bodies of a pair of paired functions with bodies which adhere to the functional congruence (3.1) or more precisely, its version that includes global variables.

Let $\langle f, g\rangle \in map_f$ be functions that we wish to make uninterpreted, and assume that f is on side 0. Each call to f is characterized by a tuple of input values and a tuple of output values. Note that the inputs include global variables that are read in the body of this function or any of its descendants. Similarly, the outputs include, in addition to the return value and values written to variables that were passed to the function by reference, the global variables that are written to in the body of this function and any of its descendants. The output values on side 0 are chosen nondeterministically. We define a structure UF-struct that holds these two tuples. Then, if there are n calls to f in side 0 (recall that this is done in the context of loop-free and recursion-free programs), we declare an array UF-array of size n and type UF-struct, and update it accordingly.

The implementation of g in side 1 is as follows. g scans the array UF-array: if g's inputs are equivalent to the inputs in entry i of UF-array, for $0 \leq i < n$, then g's outputs are assigned the values of the outputs in the same entry i. Otherwise they are assigned nondeterministic values.

A formalization of this method requires the following definition:

Definition 22 (Extended prototype)**.** An *extended prototype* of a function is a tuple consisting of:

- the list of types corresponding to the formal arguments,
- the type of the returned value,
- the sets of global variables which are read and written by the function and its successors in the call graph.

Two extended prototypes are said to be equivalent if they have the same list of types of formal arguments, the same type of returned value, and there is a bijection $\mathcal{B}$ between the globals of both sides, such that if globals $g_1, g_2 \in \mathcal{B}$ then they are paired.

Using this definition we reformulate the uninterpreted function condition (congruence). Let f and f' be two functions with the same extended prototype which are replaced by some uninterpreted function H. Let I and I' respectively be the tuples of their input arguments, O and O' be the tuples of their output arguments, G_r and G'_r be the sets of global variables they read from, G_w and G'_w be the sets of global variables write to, v and v' be their return values. We assume that each argument and global variable is paired to its counterpart. Then the congruence condition between them is:

$$(\bigvee_{a \in I} a = a' \wedge \bigvee_{g_r \in G_r} g_r = g'_r) \longrightarrow (v = v' \wedge \bigvee_{o \in O} o = o' \wedge \bigvee_{g_w \in G_w} g_w = g'_w) \qquad (9.2)$$

Following is an example of an implementation of this behavior: These are the two versions of the recursive GCD function:

```
unsigned gcd_rec(unsigned a, unsigned b)
{
    unsigned g;
    if (b == 0)
        g = a;
    else  {
        a = a % b;
        g = gcd_rec(b, a);
    }
    return g;
}
```

```
unsigned gcd_rec(unsigned x, unsigned y)
{
    unsigned z;
    z = x;
    if (y > 0)
        z = gcd_rec(y, z % y);
    }
    return z;
}
```

Figure 21: Two recursive C functions to calculate GCD.

Following is the implementation of the uninterpreted function that replaces them. The UF-struct, the UF-array and the index counter in the UF-array for each side:

```
 typedef struct
rv_UF_gcd_rec_struct_tag {
  unsigned  in_a;
  unsigned  in_b;
  unsigned  out_retval;
} rv_UF_gcd_rec_struct;

rv_UF_gcd_rec_struct rv_UF_gcd_rec_array[UF_ARRAY_LEN];

int rv_UF_gcd_rec_count[2] = {0,0};
```

The code which implements the uninterpreted function on side 0:

```
 unsigned  rvs0_gcd_rec(unsigned  a, unsigned  b) {
  unsigned  retval;

  /* check UF-array bound: */
  assert(rv_UF_gcd_rec_count[0] < UF_ARRAY_LEN);

  /* save values of input arguments and globals: */
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].in_a = a;
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].in_b = b;

  /* generate and save values of return, output arguments and globals: */
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].out_retval
    = retval = (unsigned )nondet_int();
  rv_UF_gcd_rec_count[0]++;

  return retval;
}
```

The code which implements the uninterpreted function on side 1:

```
 unsigned  rvs1_gcd_rec(unsigned  a, unsigned  b) {
```

```
  unsigned  retval;

  _Bool found = 0;
  _Bool equal = 1;
  int rv_uf_ind = rv_UF_gcd_rec_count[0]-1;
  for(; rv_uf_ind >= 0; rv_uf_ind--) {
    equal = (rv_UF_gcd_rec_array[rv_uf_ind].in_a == a);
    equal = equal && (rv_UF_gcd_rec_array[rv_uf_ind].in_b == b);
    if( equal ) {
      found = 1;
      break;
    }
  }

  if( found ) { /* input values were found among saved values */
    retval = rv_UF_gcd_rec_array[rv_uf_ind].out_retval;
  } else {
    retval = (unsigned )nondet_int();
  }
  rv_UF_gcd_rec_count[1]++;

  return retval;
}
```

Note the function name prefixes "rvs0_" and "rvs1_". They are needed as the two functions are placed together in the same check-block.

To increase the efficiency of the decision procedure we use several techniques which decrease the actual size of the possible executions of each check-block. A proper implementation of an uninterpreted function should compare each tuple of call inputs to all the call input tuples on both sides. Thus, if the number of uninterpreted function calls in an execution is C, we will need $\Omega(C^2)$ tuple compares to implement the uninterpreted function behavior. RVT allows the user to attempt a simpler proof, in which two simplifying assumptions are made. First, that the programmer does not call the same function twice with exactly the same inputs in a single execution of

the caller's body. Instead, he/she will probably keep the result in a local variable and use it twice. Therefore, RVT compares the input tuples only between sides, in the uninterpreted function code of side 1. Also, even if some function is called with the same inputs twice on one side it is improbable that the proof of equivalence depends on this. A similar optimization was made in [31], in the context of translation validation.

The second assumption is that as we compare programs which should be similar, the order of the calls to the uninterpreted function is more or less the same. In other words, we assume that if an input tuple t appeared at the k-th call to uninterpreted function F on side 0, and if the currently checked related subprograms are equivalent, then the tuple t should appear in some k'-call on side 1, such that $k - d < k' < k + d$ for some small d. Therefore we only compare each tuple encountered during execution of side 1 of the check-block with $2d - 1$ tuples on the other side. We call this value d the *look-back* value. The user may set the value by the "-lb" command line option.

This practice effectively reduces the number of tuple comparisons to $O(C \cdot d)$. Note that the above simplifications may only decrease the completeness of RVT but conserves its soundness.

Bodyless functions. Many times not all the executable code of the program is available for comparison. Some of the code is implemented in standard libraries and is not available to the user. Also, other code may be implemented in separate modules which were not changed between the two versions of the program. We call *bodyless functions* the functions whose body is not included in the compared code (only their prototype is included). We assume that the implementation of all bodyless functions were not changed between the compared versions. Therefore, we can assume that they are equivalent on both sides and we can replace them with uninterpreted functions. Note that such treatment is only good if the user knows that no bodyless function accesses global variables, as without the code of the function we can not collect the global variables that it or its descendants access. If the user is not sure that all bodyless functions were not changed, he/she can use the "-noextufs" flag of the tool

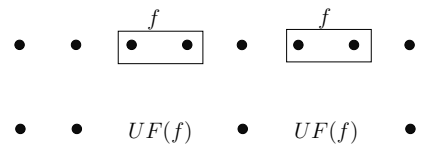

Figure 22: Each dot represents writing to the channel. The top drawing shows a possible sequence of such values written to a channel in a given function. Some of the values are written within the callee function f. After replacing f with an uninterpreted function $UF(f)$, we should check that f is called in the same location in this series of values as its counterpart on the other side.

to override the replacement. In such a case he/she must supply at least stub code to all functions reachable in the call graphs from the `main()` functions.

9.4.2 Call-sequence and call-output-sequence equivalence

We now consider the problem of proving *call-sequence equivalence*, which is needed when checking the equivalence of sequences of checkpoints as explained in the end of Sect. 9.3.3. A simple modification of this check gives us also *call-output-sequence equivalence*, which, recall, is needed for proving rule (REACT-EQ) (see the definition of the predicate call-output-seq-equiv in Sect. 7.1).

Here we consider the former problem. As explained in Sect. 9.3.3, if callees $\langle f, g \rangle \in map_f$ that are replaced with uninterpreted functions contain check points, we should consider the order in which they are called. In other words, rather than only checking that the series of values written to a given channel is the same on both sides, we should instead check that the series of such values *and calls to paired uninterpreted functions* is the same. This mixed series (values and function calls) is illustrated in Fig. 22. By this we rely on the fact that inside the body of f and g and their descendants the order of writes to the channels was already verified when f and g were proven equivalent. The generalization of this solution to multiple such pairs of callees that were replaced with uninterpreted functions is straightforward.

We can implement these checks right before each call to an uninterpreted function.

In RVT these checks are in fact implemented inside the code that simulates uninterpreted functions, as was explained in Sect. 9.4.1. We add to the code implementing the uninterpreted functions of the callees f and g code that inserts a nondeterministic value to the same channel(s) which are accessed by their bodies or their descendants. The same value is saved as part of the UF-struct on side 0 and is retrieved from it on side 1 and passed to the same channel on side 1. In this way this nondeterministic value becomes part of the output values tuple in the UF-struct.[17] Now, if these nondeterministic values appear at the same places in the channels on both sides, then the callees that generated them were called under the same conditions (guards) and in the same order. Each nondeterministic value can be thought of as a signature of the call during which it was generated. In this way, we add the signatures of the calls to the channel sequences on both sides making them effectively call-output-sequences.

The same mechanism is added to uninterpreted functions which replace functions that read inputs from some input device or their descendants read inputs. Such uninterpreted functions should write non-deterministic values to all channels which are included in the check-block.

When checking *call-output-sequence equivalence* we use a special channel for recording output values, in the same way.

Example 11. *The following code implements an uninterpreted function with code for checking call-sequence equivalence as explained above. It is the code for the gcd_rec() function (Fig. 21), assuming it contains a check-point, which is associated with the default channel. We add to the UF-struct an "unsigned char nondet_seq_equiv;" which will keep the non-deterministic value. The new uninterpreted function code for sides 0 and 1 is:*

```
 unsigned rvs0_gcd_rec(unsigned  a, unsigned b)
{
  unsigned  retval;
  unsigned  char nondet_seq_equiv;
```

[17] RVT represents this nondeterministic value with a single byte, as the number of expected function calls in the related subprograms is expected to be smaller than 256.

```
  /* check UF-array bound: */
  assert(rv_UF_gcd_rec_count[0] < UF_ARRAY_LEN);

  /* save values of input arguments and globals: */
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].in_a = a;
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].in_b = b;

  /* generate and save values of output arguments and globals: */
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].out_retval
    = retval = (unsigned )nondet_int();
  rv_UF_gcd_rec_count[0]++;

  nondet_seq_equiv = (unsigned )nondet_int();
  rv_UF_gcd_rec_array[rv_UF_gcd_rec_count[0]].nondet_seq_equiv
    = nondet_seq_equiv;
  RVSAVE(&RV_DEF_CHANNEL,1,nondet_seq_equiv);

  return retval;
}

unsigned  rvs1_gcd_rec(unsigned  a, unsigned  b)
{
  unsigned  retval;
  unsigned  char nondet_seq_equiv;

  _Bool found = 0;
  _Bool equal = 1;
  int rv_uf_ind = rv_UF_gcd_rec_count[1];
  if( rv_uf_ind > rv_UF_gcd_rec_count[0]-1 )
    rv_uf_ind = rv_UF_gcd_rec_count[0]-1;
  {
    equal = (rv_UF_gcd_rec_array[rv_uf_ind].in_a == a);
    equal = equal && (rv_UF_gcd_rec_array[rv_uf_ind].in_b == b);
    if( equal ) {
```

```
      found = 1;
    }
  }

  if( found ) /* input values were found among the saved values */
  {
    retval = rv_UF_gcd_rec_array[rv_uf_ind].out_retval;
    nondet_seq_equiv = rv_UF_gcd_rec_array[rv_uf_ind].nondet_seq_equiv;
    RVCHECK(&RV_DEF_CHANNEL,1,nondet_seq_equiv);

  } else {
    /* Assert call-sequence equivalence: call inputs must be the same. */
    assert( 0 );
  }
  rv_UF_gcd_rec_count[1]++;

  return retval;
}
```

RVT adds generation and saving to the default channel of the nondet_seq_equiv value to the code of side 0, and its retrieval and value check to the code of side 1. Note that in the code of side 1 above there is no loop and the only UF-struct we look at in the UF-array is the one at the current index of "rv_UF_gcd_rec_count[1]". This is because the sequence of calls to the uninterpreted function must be exactly the same on both sides and thus the index of the related calls is the same.

9.4.3 Reach equivalence

We now consider the problem of proving *reach equivalence*, which is necessary for rule (M-TERM). To prove reach-equivalence for two isolated related subprograms, we need to show that for argument-equivalent executions of the related subprograms, for every call to a function f in the execution on side 0 there is a call to a function g in the execution on side 1 (and vice-versa) such that $\langle f, g\rangle \in map_f$ and the calls are with equal arguments. Our semantic-check mechanism for *computational* equivalence, as

described in Sect. 9.3.3 in the context of checking rule (PROC-P-EQ), already enforces argument-equivalent executions. For reach-equivalence check, we should add to this mechanism:

1. an assertion that any input tuple (i.e., the arguments of the callee) on side 1 is found in the UF-array, and

2. a check that for every UF-struct of the UF-array its input tuple was found to be equivalent to some input tuple on side 1.

These checks can be implemented by the corresponding changes in the uninterpreted function code of side 1:

1. add assertion which fails if the input tuple was not found by the "for" loop, and

2. add a Boolean flag (called "reach_equiv_flag") to the UF-struct which will be false by default, let the "for" loop go over all the UF-array and set to true the flag of each UF-struct whose input tuple is equal to the received input tuple. At the end of the check-block code add a check that all these reach_equiv_flags were set to true.

Note that as the check is performed in the code that implements an uninterpreted function the calls we compare are to paired functions. This satisfies the remaining condition of reach-equivalence.

Example 12. *Below is an example of an implementation of the reach-equivalence check for the gcd_rec() function that we considered in Sect. 9.4.2. We add to the UF-struct (rv_UF_gcd_rec_struct) another Boolean field "reach_equiv_flag;" which is initialized to 0 (false) when side 0 fills the UF-struct. The change in the code of side 1 is:*

```
unsigned  rvs1_gcd_rec(unsigned  a, unsigned  b) {
 unsigned  retval;
```

```
  _Bool found = 0;
  _Bool equal = 1;
  int rv_uf_ind = rv_UF_gcd_rec_count[0]-1;
  for(; rv_uf_ind >= 0; rv_uf_ind--) {
    equal = (rv_UF_gcd_rec_array[rv_uf_ind].in_a == a);
    equal = equal && (rv_UF_gcd_rec_array[rv_uf_ind].in_b == b);
    if( equal ) {
      found = 1;
      rv_UF_gcd_rec_array[rv_uf_ind].reach_equiv_flag = 1;
    }
  }

  if( found ) { /* input values were found among the saved values */
    retval = rv_UF_gcd_rec_array[rv_uf_ind].out_retval;
  } else {
    /* Assert reach-equivalence:
       side 1 inputs must appear on side 0. */
    assert( 0 );
  }
  rv_UF_gcd_rec_count[1]++;

  return retval;
}
```

The "assert(0)" at the lower part of function rvs1_gcd_rec() checks that all call configuration of side 1 were also reached by side 0.

Additionally, in the main code of the check-block we add a check that all the reach_equiv_flag bits are set to 1 (true) in all used UF-structs of all uninterpreted functions which appear in this check-block. This checks that all function calls reached by side 0 were also reached by side 1, and with the same argument values.

Recall that if all functions passed the computational equivalence check (as described in Sect. 9.3.3) and the reach-equivalence check as presented here, then we can conclude that the programs are mutually terminating.

9.5 Experiments

We have tested RVT on several synthetic and limited-size industrial programs and attempted to prove equivalent different versions of these programs. The programs are described below.

9.5.1 Synthetic programs

The small synthetic programs which we have checked:

- The GCD program on which many examples in this thesis are based. It calculates the greatest common divisor of two integers. The program was tested in two versions: with a simple function containing a loop and a version with only a recursive function.
- The simple_loops test that contains four different functions with loops that calculate power, sum of factors and sum of hexadecimal digits on integer numbers.
- A C version of the HPcalc test that is used as an example in figure 14.
- MSCC1 is a simple program that tests the execution of RVT on programs with MSCCs.
- cr1 and cr2 test RVT on functions which perform evaluation of a numeric expression represented as a dynamically-allocated structure.

We compared each of these programs with small modifications thereof. Some changes kept the programs equivalent and others made them different. In all cases, when the programs were equivalent, RVT proved their partial equivalence within seconds up to a minute. When checking the same programs for k-equivalence, it took RVT several minutes up to several hours for k up to 5. For higher values of k, the test many times could not prove k-equivalence even after more than 10 hours. In the case of nonequivalent programs, both the k-equivalence and the semantic-checks found a separating execution of the programs in several seconds.

Recall that in the process of semantic checks, paired functions that cannot be proven equivalent, are (logically) inlined. Our experience was that in such cases the proof becomes too hard: the decision procedure runs for hours or even fails to reach a decision at all. Executing k-equivalence with a low k value instead would reveal the nonequivalence much faster.

Interestingly, we have found in the examples above that if we isolate 'hard' operators (i.e., operators that their Boolean representation burden the SAT solver) such as multiplication (*), division (/) and modulo (%) into separate functions, this many times solves the computational problem. The reason is that these separate functions are replaced by uninterpreted functions, and hence the execution time of semantic-checks and especially k-equivalence on these tests decrease dramatically. For example, in the case of k-equivalence with $k = 6$ on equivalent programs it would decrease the time of simple_loops from tens of hours to less than a minute. This means that a big part of the k-equivalence test complexity lies in such operators.

Bigger programs that do not use arrays are very rare. Since we did not finish the implementation of a solution to the case of arrays as arguments to functions, we could not find suitable large programs to check.[18] Instead, we developed an automatic program generator (called "gen_prog") to generate sizable random programs in several close versions and then test them using RVT. The programs are generated by randomly building a parse tree of a program. These programs include many function calls, including recursive and mutually recursive calls. The user specifies the probability to generate each type of variable, block, or operator. Variables can be global, local or formal arguments of functions. Types can be basic C types, structures or pointers to such types, but not arrays.

Also, small differences in versions are introduced in random places by enclosing some statements in "#ifdef RV_DIFF" directives. This way, using the C compiler preprocessor (CPP) and choosing different defines in different times one can generate several close but not identical versions. We used gen_prog to generate random yet

[18] RVT can handle a restricted case in which the arrays are 'read-only'.

executable C programs with up to 20 functions and thousands of lines of code.

When the random versions are equivalent, RVT proves them to be partially-equivalent relatively fast: from seconds to half an hour. On non-equivalent versions, on the other hand, attempts to prove partial equivalence may run for many hours or run out of memory. Retreating to k-equivalence sometimes solves the problem, but not always as the programs are too big even for low values of k.

9.5.2 Industrial programs

We also tested our tool on several limited-size industrial programs. These are:

TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircraft. We used a 300-line fragment of this program that was also used in [16].

MicroC/OS The core of MicroC/OS which is a low-cost priority-based preemptive real time multitasking operating system kernel for microprocessors, written mainly in C. The kernel is mainly used in embedded systems. The program is about 3000 lines long.

Matlab examples Parts of engine-fuel-injection simulation in Matlab which was generated in C from engine controller models. The tested parts contain several hundreds lines of code and use read-only arrays.

All these tests exhibit the same behavior as the syntactic tests above. For equivalent programs, semantic-checks were very fast, proving equivalence in minutes. In the case of non-equivalent programs after initial failure to prove equivalence using semantic-checks, k-equivalence would sometimes succeed to show a difference between the two versions of the program, but mostly on variations of the TCAS example, which is the smaller of the three programs listed above.

10 Future work and summary

We divide the list of future work items to implementation issues and further research directions.

- Implementation issues.

  Arrays. RVT currently does not support programs with arrays (unless these are read-only arrays), which most industrial programs use.

  Rules (M-TERM) **and** (REACT-EQ)**.** The implementation of these two rules in RVT is not finished.

- Future research directions.

  Checking real code revisions. Once RVT is capable of proving significant industrial programs, it will be interesting to see whether it is capable of proving the equivalence of two versions of code that reflect a real change (e.g., checking consecutive revisions in a CVS archive). A related software-engineering question is to examine the ideal gap between programs to apply regression verification (a question that also applies to regression testing).

  Characterizing the strength of the inference system. Checking the strength of the rules with respect to real code changes - which types of correct code transformations that are in practical use can these rules prove.

  Integrate additional static analysis techniques. For example, it could be that a large part of the code cannot affect the equivalence of two procedures that we wish to prove equivalent. Slicing of the program can remove the irrelevant code in such a case.

Summary We started the introduction by mentioning Tony Hoare's grand challenge, namely that of building a verifying compiler, and by mentioning that proving equivalence is a grand challenge in its own right, although an easier one. In this thesis we started exploring this direction in the context of real programs written in C.

The main contributions of the thesis can be summarized as follows:

1. Introducing various notions of equivalence (some of which were already considered in the past).

2. Introducing inference rules for proving equivalence of recursive programs, according to the various equivalence notions. Each rule is accompanied by a proof of soundness.

3. Introducing a method for an automatic, incremental proof, based on isolating functions from their callees and abstracting them with uninterpreted functions. This method keeps the verification conditions decidable and small relative to the size of the input programs. Our algorithm initially compares functions syntactically which, under certain conditions, circumvents the need to call the decision procedure. Such preliminary syntactic checks simplify the process significantly in the case targeted in this work, namely of comparing programs that have large parts of equal code. It also helps meeting our goal, as stated in the introduction, of attempting to keep the complexity sensitive to the changes rather than to the original size of the compared programs. Only if the syntactic checks are unable to conclude that functions are equivalent, it invokes the more expensive semantic checks based on the rules and the decision procedure CBMC.

4. Developing methods for applying the above rules and algorithms to C programs.

Acknowledgments

I would like to thank my supervisor, Dr. Ofer Strichman for his guidance and help. I would like to thank Mooly Sagiv, Shmuel Katz and especially Daniel Kroening for helpful discussions and good advice.

A Formal definitions for Section 7

Definition 23 (Input/output subsequences of a subcomputation)**.**
Let π' be a subcomputation of a computation π. If π' is finite then the *input subsequence* of π' is the prefix sequence Q_{IN} that satisfies $\text{first}[\pi'].\overline{R} = Q_{IN} \cdot \text{last}[\pi'].\overline{R}$ and the *output subsequence* of π' is the tail sequence Q_{OUT} that satisfies $\text{first}[\pi'].\overline{W} \cdot Q_{OUT} = \text{last}[\pi'].\overline{W}$. If π' is infinite then the *input subsequence* of π' is simply $\text{first}[\pi'].\overline{R}$ and the *output subsequence* of π' is the tail sequence Q_{OUT} that satisfies $\text{first}[\pi'].\overline{W} \cdot Q_{OUT} = \text{OutSeq}[\pi]$. $\diamond$

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

We mark the input subsequence of π' 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.

Definition 24 (Reactive equivalence of two procedures)**.**
Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map_f$, if for every two subcomputations π'_1 and π'_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*. $\diamond$

Denote by *reactive-equiv*(F, G) the fact that F and G are reactively equivalent.

Definition 25 (Return-values equivalence of two reactive procedures)**.**
If for every two finite subcomputations π'_1 and π'_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*. $\diamond$

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

Denote by *return-values-equiv*(F, G) the fact that F and G are return-value equivalent.

Definition 26 (Inputs-suffix equivalence of two reactive procedures)**.**
If for every two finite subcomputations π'_1 and π'_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 27 (Output configuration)**.** A configuration C is an *output configuration* if current-label$[C]$ = before[**output**(e)]. ◇

Definition 28 (Call and output sequence of a subcomputation)**.** The *call and output sequence* of a subcomputation π' contains all the call and output configurations in π' in the order in which they appear in π'. ◇

Definition 29 (Call and output sequence equivalence between subcomputations)**.** Finite subcomputation π'_1 and π'_2 from some levels are *call and output sequence equivalent* if the call-and-output-sequences CC_1 of π'_1 and CC_2 of π'_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** $p_2(\overline{e}_2; \overline{x}_2)$],
   - $\langle p_1, p_2 \rangle \in map_f$, 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 30 (Call and output sequence equivalence of two procedures)**.** Given two procedures $F \in Proc[P_1]$ and $G \in Proc[P_2]$ such that $\langle F, G\rangle \in map_f$, if for every two finite subcomputations π_1' and π_2' that are input equivalent with respect to F and G it holds that π_1' and π_2' are call and output sequence equivalent, then F and G are *call and output sequence equivalent.* ◇

Denote by *call-output-seq-equiv*(F, G) the fact that F and G are call-sequence equivalent.

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 function, 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 [12] (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).

C Implementation issues

C.1 Converting loops to recursive functions

Algorithms 1 and 2 treat the compared programs as a collection of functions without loops. We therefore need to replace all loops in the programs by recursive functions. We now present the schema for treating "for" loops. We replace the following schematic code:

```
<return-type> <original-function>(...) {
  . . .
  for( <INIT>; <COND>; <UPDATE>) {
      <BODY>
  }
  . . .
}
```

with the following code:

```
<return-type> <original-function>(...) {
  . . .
  <INIT>;
  <for-loop-recursive-function>(<addresses-of-loop-variables>);
  . . .
}

<for-loop-recursive-function>(<pointers-to-loop-variables>)
{
  if( !<COND> )
    return;

  <BODY>

  continue_label:

  <UPDATE>;

  /* the recursive call to the next loop iteration: */
  <for-loop-recursive-function>(<pointers-to-loop-variables>);

  break_label:
}
```

In the above schema, ⟨INIT⟩, ⟨COND⟩, ⟨UPDATE⟩ and ⟨BODY⟩ represent various code blocks of which the loop is comprised. We cut the loop code (except the ⟨INIT⟩ part) out of the original function and replace it by a call to the function ⟨for-loop-recursive-function⟩ which will implement the iterations of this loop recursively. As the original loop, this function evaluates ⟨COND⟩ and proceeds into the loop body only if it is true. After we execute the body of the loop, we execute the ⟨UPDATE⟩ block and call recursively to the function to possibly execute subsequent loop iterations. We pass the variables used in the loop by address to this function. Therefore we need to replace each access to a variable in the loop by an access to a dereference of its respective pointer.

Note that the ⟨BODY⟩ code may contain "break" and "continue" statements. We replace these statements by "goto break_label" or "goto continue_label" respectively. These labels are placed in the right places to emulate the behavior of the original loop on these statements.

The "while" and the "do-while" loops are treated the same way as "for" loops with the following minor changes. As they contain no ⟨INIT⟩ or ⟨UPDATE⟩ code, no such code appears in the recursive implementation. Also, in the case of "do-while" loop, the "if(!⟨COND⟩) return;" statement appears after the ⟨BODY⟩ and before the "continue_label:" instead of being at the beginning of the new function.

Following is a simple loop-to-recursion example. We replace the following code:

```
struct Str1 {
  int   key;
  long  data;
  struct Str1 *left, *right;
};

long find(struct Str1* pstr, int key)
{
    while( pstr && pstr->key != key ) {
        if( pstr->key < key )
            pstr = pstr->right;
        else pstr = pstr->left;
    }
    return pstr;
}
```

with the next one (the definition of "Str1" stays the same):

```
long find(struct Str1* pstr, int key) {
    find_rec_loop0( &pstr, key );

    return pstr;
```

```
}

void  find_rec_loop0(struct Str1** ppstr, int key)
{
    if( !( *ppstr && (*ppstr)->key != key ) )
        return;

    if( (*ppstr)->key < key )
        *ppstr = (*ppstr)->right;
    else *ppstr = (*ppstr)->left;

    find_rec_loop0( ppstr, key );
}
```

Note that the whole body of the loop was replaced by the find_rec_loop0() recursive function. The "if" statement which guards the entrance to the function contains the negated loop condition.

Here is a more elaborate example which contains also "continue" and "break" statements. We replace the following code:

```
int sum(int *pa, int len) {
    int i,s;

    for(i = 0, s = 0;  i  <  len;  i++)  {
        if( pa[i] < 0 )
           continue;
        if( pa[i] == 0 )
           break;
        s += pa[i];
    }

    return s;
}
```

with the following one:

```
int sum_rec(int *pa, int len) {
  int i,s;

  i = 0, s = 0;
  sum_rec_loop0(i, &s, len, pa);

  return s;
}

void sum_rec_loop0(int i, int *ps,
                   int len, int *pa)
{
    if( !(i < len) )
        return;

    if( pa[i] < 0 )
        goto l_continue;
    if( pa[i] == 0 )
        goto l_break;
    *ps += pa[i];

  l_continue:
    i++;
    sum_rec_loop0(i, ps, len, pa);
  l_break:;
}
```

Both versions of "sum()" return the sum of all positive elements of the array "pa" till the first 0 element in the array. Note that in the original code the "i" variable is used only inside the loop. Therefore, it is passed to sum_rec_loop0() by value. In contrast, "s" is used outside the loop and therefore is passed to sum_rec_loop0() by reference.

Note that in the ⟨BODY⟩ code, "return" statements and "goto" statements may appear. These statements may cause the execution to exit the loop. Accordingly,

when we transform the loop to a separate function, we need a way to perform these gotos and returns as if they were executed from the original function. For the "return" statement we add to the original function a placeholder for the returned value and pass its address to the loop recursive function. Different loop-exit statements may have various target locations: various gotos labels which are placed outside the loop body, the program location immediately after the loop, or out of the function which contains the loop (by a "return" statement). To distinguish between those locations the loop recursive function returns a "loop termination code" value which signals to the calling function what is the target of the exit. A code value of 0 (LTC_NORMAL) means a usual exit of the loop: because ⟨COND⟩ is false or because of a "break" statement. A code of LTC_RETURN means that the calling function should return with the value specified by the return value placeholder. Code values 1, 2, 3, etc. result in goto jumps to the respective labels. Here is an example of such a transformation:

Here is a loop-to-recursion example with "return" and "goto" labels. We replace the following code:

```
int sum(int *pa, int len) {
    int i,s;

    for(i = 0, s = 0;  i  <  len;  i++)  {
        if( pa[i] < 0 )
           goto lab_negative_value;
        if( pa[i] == 0 )
           goto lab_zero_value;
        s += pa[i];
        if( s > 1000 )
          return 1000;
    }
    return s;

  lab_negative_value:
    report("Negative value at index:", i);
    return -1;
```

```
  lab_zero_value:
    report("Zero value at index:", i);
    return 0;
}
```

with the next one:

```
int sum(int *pa, int len) {
    int  rvretval;
    int  i,s;

    i = 0 , s = 0;
    switch(sum_rec_loop0(&i,&s,len,pa,&rvretval))
    {
      case 1:
        goto lab_negative_value;
      case 2:
        goto lab_zero_value;
      case LTC_RETURN:
        return rvretval;
      case 0:
        break;
    }
    return s;

lab_negative_value:
    report("Negative value at index:",i);
    return -1;

lab_zero_value:
    report("Zero value at index:",i);
    return 0;
}

unsigned char sum_rec_loop0(int *pi, int *ps,
                            int len, int *pa, int *prvretval)
{
```

```
    if( !(*pi < len) )
        return 0;
    if (pa[*pi] < 0)
        return 1;
    if (pa[*pi] == 0)
        return 2;
    *ps += pa[*pi];
    if (*ps > 1000)
    {
        *prvretval = 1000;
        return LTC_RETURN;
    }

    (*pi)++;
    return sum_rec_loop0(pi,ps,len,pa,prvretval);

    return 0;
}
```

Note that the loop-to-recursion mechanism described here cannot perform goto jumps into loop bodies. Though the implementation of these jumps is possible in principal, we did not implement them as they are extremely rare and are considered a bad programming practice.

Bibliography

[1] Tamarah Arons, Elad Elster, Limor Fix, Sela MadorHaim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, , and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In K. Etessami and S. Rajamani, editors, *Proc.* 17^{th} *Intl. Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lect. Notes in Comp. Sci.*, Edinburgh, July 2005. Springer-Verlag.

[2] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001. SLAM.

[3] Luc Bouge and David Cachera. A logical framework to prove properties of alpha programs (revised version). Technical Report RR-3177, 1997.

[4] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[5] Sorin Craciunescu. Proving the equivalence of clp programs. In *ICLP*, pages 287–301, 2002.

[6] David W. Currie, Alan J. Hu, and Sreeranga P. Rajan. Automatic formal verification of dsp software. In *DAC*, pages 130–135, 2000.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control

dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[8] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of SPIN2000*. Springer-Verlag, 2000.

[9] Xiushan Feng and Alan J. Hu. Automatic formal verification for scheduled vliw code. In *LCTES-SCOPES*, pages 85–92, 2002.

[10] Xiushan Feng and Alan J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT*, pages 307–316, 2005.

[11] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[12] Martin Fowler. http://www.refactoring.com.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] Nissim Francez. *Program Verification*. Addison-Wesley, 1993.

[15] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. (submitted to Acta-Informatica), 2008.

[16] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *CAV*, pages 453–456, 2004.

[17] Malek Haroud and Armin Biere. SDL versus C equivalence checking. In *SDL Forum*, pages 323–338, 2005.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002. BLAST.

[19] C.A.R Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[20] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003.

[21] Tony Hoare and Jay Misra. Verified software: theories, tools, experiments vision of a grand challenge project. In *VSTTE*, July 2005. Also a Microsoft technical report MSR-TR-2006-117.

[22] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[23] D. C. Luckham, M. R. Park, and M. S. Paterson. On formalized computer programs. *Jour. Comp. and System Sciences*, 4(3), June 1970.

[24] Panagiotis Manolios. *Computer-Aided Reasoning: ACL2 Case Studies*, chapter Mu-Calculus Model-Checking, pages 93–111. Kluwer Academic Publishers, 2000.

[25] Panagiotis Manolios and Matt Kaufmann. Adding a total order to acl2. In *The Third International Workshop on the ACL2 Theorem Prover*, 2002.

[26] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 2006. to appear.

[27] G. C. Necula. Translation validation for an optimizing compiler. In *In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, june 2000.

[28] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 235–246. Springer-Verlag, 1998.

[29] A. Pnueli, M. Siegel, and O. Shtrichman. *Translation Validation: From SIGNAL to C*, volume 1710 of *LNCS State-of-the-Art Survey*, pages 231–255. Springer-Verlag, 1999.

[30] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. Technical report, Sacres and Dept. of Comp. Sci., Weizmann Institute, Apr. 1997.

[31] Amir Pnueli and Ofer Strichman. Reduced functional consistency of uninterpreted functions. In *Pragmatics of Decison Procedures for Automated Reasoning (PDPAR'05)*, number 898 in Electronic Notes in Computer Science, 2005.

[32] Amir Pnueli and Ganna Zaks. Validation of interprocedural optimizations. In *7th Int. Workshop: Compiler Optimization Meets Compiler Verification (COCV'08), Satellite workshop of ETAPS'08*, 2008.

[33] Terrence W. Pratt. Kernel equivalence of programs and proving kernel equivalence and correctness by test cases. *International Joint Conference on Artificial Intelligence*, 1971.

[34] Robert Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583 – 585, July 1978.

[35] D. Tsichritzis. The equivalence problem of simple programs. *J. ACM*, 17(4):729–738, 1970.

[36] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.