Program Equivalence by Circular Reasoning

Dorel Lucanu, Vlad Rusu

Program Equivalence by Circular Reasoning

Dorel Lucanu, Vlad Rusu

RESEARCH REPORT
N° 8116
octobre 2012
Project-Team Dart

ISSN 0249-6399   ISRN INRIA/RR--8116--FR+ENG

Program Equivalence by Circular Reasoning

Dorel Lucanu [*], Vlad Rusu[†]

Project-Team Dart

Research Report n° 8116 — octobre 2012 — 22 pages

Abstract: We propose a logic and a deductive system for stating and automatically proving the equivalence of programs in deterministic languages having a rewriting-based operational semantics. The deductive system is circular in nature and is proved sound and weakly complete; together, these results say that, when it terminates, our system correctly solves the program-equivalence problem as we state it. We show that our approach is suitable for proving the equivalence of both terminating and non-terminating programs, and also the equivalence of both concrete and symbolic programs. The latter are programs in which some statements or expressions are symbolic variables. By proving the equivalence between symbolic programs, one proves in one shot the equivalence of (possibly, infinitely) many concrete programs obtained by replacing the variables by concrete statements or expressions. We also report on a prototype implementation of the proposed deductive system in the $\mathbb{K}$ Framework.

Key-words: Program Equivalence, Circular Reasoning, $\mathbb{K}$ framework.

[*] University of Iasi, Romania
[†] Inria Lille Nord Europe

Equivalence de Programmes par Raisonnement Circulaire

Résumé : Nous proposons une logique et un système déductif pour exprimer et prouver automatiquement l'équivalence de programmes dans des langages déterministes munis de sémantiques opérationnelles définies par réécriture. Le système déductif proposé est de nature circulaire; nous démontrons qu'il est correct et faiblement complet. Ces deux résultats signifient que notre système résout correctement le problème d'équivalence de programmes tels que nous l'avons posé. Nous montrons que ce système fonctionne autant pour des programmes qui terminent que pour des programmes qui ne terminent pas. Les programmes dits symboliques, dans lesquels certaines expressions ou instructions restent non-interprétés, peuvent également être traités par notre approche. La démonstration d'une équivalence entre deux programmes symboliques revient à démontrer l'équivalence entre une infinité potentielle de programmes concrets, qui sont des instances des programmes symboliques obtenues en remplaçant les variables symboliques par des instructions ou des expressions concrètes. Enfin, nous décrivons une implémentation prototype de notre système déductif dans la $\mathbb{K}$ *framework*.

Mots-clés : Equivalence de programmes, Raisonnement circulaire, $\mathbb{K}$ *framework*.

1 Introduction

In this paper we propose a formal notion of program equivalence, together with a logic for expressing this notion, and a deductive system for automatically proving it. Programs can belong to any deterministic language whose semantics is specified by a set of rewrite rules. The equivalence is a form of weak bisimulation, allowing several instructions of one program to be matched by several instructions of the other one. The proof system is circular: its conclusions can be re-used as hypotheses in a controlled way. It is not guaranteed to terminate, but when it does terminate, our proof system correctly solves the program-equivalence as stated, thanks to its soundness and weak completeness properties. These properties are informally presented below and are formalised and proved in the paper.

The proposed framework is also suitable for proving the equivalence of *symbolic programs*. These are programs in which some expressions and/or statements are *symbolic variables*, which denote sets of concrete programs obtained by substituting the symbolic variables by concrete expressions and/or statements. Thus, by proving the equivalence between symbolic programs, one proves in just one shot the equivalence of (possibly, infinitely) many concrete programs. This has applications in the verification of certain classes of compilers/translators. Here is an example of equivalent symbolic programs. *Example:* Assume that we want to translate between a language that has `for`-loops into a language that only has `while`-loops. This amounts to translating the symbolic program in the left-hand side to the one in the right-hand side.

$$\texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}\ S\ \texttt{\}} \qquad I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do}\ \texttt{\{}\ S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+ 1 \}}$$

Their symbolic variables I, A, B, S can be matched by, respectively, any identifier (I), arithmetical expression (A, B), and program statement (S). If we prove the equivalence between these two symbolic programs (as we shall do in this paper as an illustrative example) then we also prove that every concrete instance of the `for`-loop is equivalent to its translation to a concrete `while`-loop (or vice-versa). Nonterminating programs can be considered as well, e.g. instances of the above two symbolic programs. In the rest of the paper we often refer to symbolic programs just as "programs".

A typical use of our program-equivalence framework consists in:

1. defining the operational semantics of a programming language, say, $\mathcal{L}$;

2. defining a language $\mathcal{L}^{sym}$, which extends the syntax of and semantics of $\mathcal{L}$, such that the programs in $\mathcal{L}^{sym}$ are exactly the symbolic programs of $\mathcal{L}$;

3. running our deductive system to check the equivalence of programs in $\mathcal{L}^{sym}$.

Running the deductive system amounts essentially to executing the semantics of $\mathcal{L}^{sym}$ on pairs of $\mathcal{L}^{sym}$-programs. This may lead to any of the following outcomes:

- termination with success, in which case the programs given as input to the deductive system are equivalent, due to the deductive system's *soundness*;

- termination with failure, in which case the programs given as input to the deductive system are not equivalent, due to the system's *weak completeness*;

- non-termination, in which case nothing can be concluded about equivalence.

Non-termination is inherent in any sound automatic system for proving program equivalence, because the equivalence problem is undecidable. We show, however, that our system terminates when the programs given to it as inputs terminate, and also when they do not terminate but behave in a certain regular way (by infinitely repeating pairs of so-called *observationally equivalent configurations*).

Contributions:

A logic and a proof system suitable for stating and proving the equivalence of concrete and of symbolic programs, as well as that of terminating and non-terminating ones. Programs can be written in any deterministic language that has a formal operational semantics based on term rewriting. We prove the soundness and weak completeness of our proof system, which ensure that the system correctly solves the program equivalence problem as we state it.

Related Work

An exhaustive bibliography on the program-equivalence problem is outside the scope of this paper, as this problem is actually older than the program-verification problem. Among the recent works perhaps the closest to ours is [1]. They also deal with the equivalence of parameterised programs (symbolic, in our terminology) and define equivalence in terms of bisimulation.

Their approach is, however, very different from ours. One major difference lies in the models of programs: [1] use CFGs (control flow graphs) of programs, while we use the operational semantics of languages. CFGs are more restricted, e.g., they are not well adapted to recursive or object-oriented programs, whereas operational semantics do not have these limitations. Of course, our advantage will only become apparent when we actually apply our approach to such programs.
Other closely related recent works are [2, 3, 4]. The first one targets programs that include recursive procedures, the second one exploits similarities between single-threaded programs in order to prove their equivalence, and the third one extends the latter to multi-threaded programs. They use operational semantics (of a specific language) and proof systems, and formally prove their proof system's soundness. In [2] they make a useful classification of equivalence relations used in program-equivalence research, and use these relations in their work.

However, all the relations classified in [2] are of an input/output nature: for given (sequences of) inputs, programs generate equal (sequences of) outputs and/or do not terminate. Such relations are well adapted for concrete programs with inputs and outputs, but not to symbolic programs with symbolic statements, for which a clear input-output relation may not exist. Indeed, symbolic statements may denote arbitrary concrete statements - including ones that do not perform input/output - actually, when symbolic programs are concerned, one cannot even rely on the existence of inputs and outputs. One may rely, however, on the observations of the effects of symbolic statements on the program's environment (e.g., values of variables). Our notion of weak bisimulation (up to a certain observation relation) allows this, both for finitely and for infinitely many repeated observations. We also show that some of the relations from [2] can be encoded in our relation by adding information tp the program environment.

Many works on program equivalence arise from the verification of compilation in a broad sense. At one end there is full compiler verification [5], and at the other end, the so-called translation validation, i.e., the individual verification of each compilation [6] (we only cite two of the most relevant recent works). As also observed by [1], symbolic program verification can also be used for simple compilers, in which one proves the equivalence of each basic instruction pattern from the source language with its translation in the target language. The application of this observation to the verification of a compiler (from another project we are involved in) is ongoing and will be presented in another paper.

Several other works have targeted specific classes of languages: functional [7], microcode [8], CLP [9]. In order to be less language-specific some works advocate the use of intermediate languages, such as [10], which works on the Boogie intermediate language. And finally, only some few approaches, among which [5, 8], deal with real-life language and industrial-size programs in those languages. This is in contrast to the equivalence checking of hardware circuits, which has entered the mainstream industrial practice (see, e.g., [11] for a survey on this topic).

Our proof system is inspired by that of *circular coinduction* [12], which allows one to prove equalities of data structures such as infinite streams and regular expressions. A notable difference between the present approach and [12] is that our specifications are essentially rewrite theories (meant to define the semantics of programming languages), whereas those of [12] are behavioural equational theories, a special class of equational specifications with visible and hidden sorts.

The rest of the paper is organised as follows. Section 2 presents our running example: IMP, a simple imperative language and its semantics in $\mathbb{K}$ [13]. $\mathbb{K}$ is a formal framework for defining operational semantics of programming languages.

Section 3 introduces some formal notions on language definitions, useful in defining program equivalence, and illustrates them on the $\mathbb{K}$ semantics of IMP.

Section 4 contains our proposed definition for program equivalence, and Section 5 gives the syntax and semantics of a logic capturing the chosen equivalence.

Section 6 introduces two operations on formulas of the logic (derivatives and conjunction) which are used in our circular proof system for formula validity.

The proof system itself is presented in Section 7, together with its soundness and weak completeness results. The results say that, when it terminates, the proof system correctly answers to the question of whether its input (which is a set of formulas in our program-equivalence logic) denotes equivalent programs.

In Section 8 we report on a prototype implementation of the proof system in the $\mathbb{K}$ framework. This allows one to stay within the $\mathbb{K}$ environment when proving program equivalence for languages whose semantics is also defined in $\mathbb{K}$.

The conclusion and future work are presented in Section 9. Finally, formal proofs of the results in the paper are given in an Appendix.

2 A Simple Imperative Language and its Semantics in $\mathbb{K}$

The language we are using as running example is IMP, a simple imperative language intensively used in research papers. A full $\mathbb{K}$ definition of it can be found in [13]. The syntax of IMP is described in Figure 1 and is mostly self-explained. The attribute (given as an annotation) *strict* from the syntax means the arguments of the annotated expression/statement are evaluated before the expression/statement itself is evaluated/executed. If the attribute has as arguments a list of natural numbers, then only the arguments in positions specified by the list are evaluated before the expression/statement. The *strict* attribute is actually syntactic sugar for a set of $\mathbb{K}$ rules, briefly presented later in the section.

The *configuration* of an IMP program consists of code to be executed and an enviroment mapping identifiers to integers. In $\mathbb{K}$, this is written as a nested structure of *cells*: here, a top cell `cfg`, having a cell `k` and a cell `env` (see Figure 2).

The cell `k` includes the code to be executed, represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$, meaning that first C_1 will be executed, then C_2, etc. Computation tasks are typically the evaluation of statements and expressions. The cell `env` is an environment that binds the program variables to values; such a binding is written as a multiset of bindings of the form, e.g., `a` $\mapsto 3$.

The semantics of IMP is given by a set of rules (see Figure 3) that say how the configuration evolves when the first computation task (statement or instruction) from the `k` cell is executed. The dots in a cell mean that the rest of the cell remains unchanged. Except for the conjunction and the conditional statement, the semantics of each operator and statement is described by

$$\begin{array}{ll}
\mathit{Int} & ::= \text{domain of integer numbers (including operations)}\\
\mathit{Bool} & ::= \text{domain of boolean constants (including operations)}\\
\mathit{Id} & ::= \text{domain of identifiers}
\end{array}$$

$$\begin{array}{ll}
\mathit{AExp} ::= \mathit{Int} \mid \mathit{Id} & \mathit{BExp} ::= \mathit{Bool}\\
\quad \mid \mathit{AExp}\ \texttt{/}\ \mathit{AExp}\ [\text{strict}] & \quad \mid \mathit{AExp}\ \texttt{<=}\ \mathit{AExp}\ [\text{strict}]\\
\quad \mid \mathit{AExp}\ \texttt{*}\ \mathit{AExp}\ [\text{strict}] & \quad \mid \texttt{not}\ \mathit{BExp}\ [\text{strict}]\\
\quad \mid \mathit{AExp}\ \texttt{+}\ \mathit{AExp}\ [\text{strict}] & \quad \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp}\ [\text{strict}(1)]\\
\quad \mid (\mathit{AExp}) & \quad \mid (\mathit{BExp})
\end{array}$$

$$\begin{array}{ll}
\mathit{Stmt} ::= \texttt{skip} \mid \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt} & \mid \texttt{\{}\ \mathit{Stmt}\ \texttt{\}}\\
\quad \mid \mathit{Id}\ \texttt{=}\ \mathit{AExp} & \mid \texttt{while}\ \mathit{BExp}\ \texttt{do}\ \mathit{Stmt}\\
\quad \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Stmt} & \mid \texttt{for}\ \mathit{Id}\ \texttt{from}\ \mathit{AExp}\ \texttt{to}\ \mathit{AExp}\\
\quad\ \ \texttt{else}\ \mathit{Stmt}\ [\text{strict}(1)] & \quad \texttt{do}\ \mathit{Stmt}\ [\text{strict}(2,3)]
\end{array}$$

$$\mathit{Code} ::= \mathit{Id} \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{AExp} \mid \mathit{BExp} \mid \mathit{Stmt} \mid \mathit{Code} \curvearrowright \mathit{Code}$$

Figure 1: $\mathbb{K}$ Syntax of IMP

$$\mathit{Cfg} \quad ::= \langle\langle \mathit{Code}\rangle_{\mathsf{k}} \langle \mathit{Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

Figure 2: $\mathbb{K}$ Configuration of IMP

exactly one rule.

In Figure 3, the operations `lookup`: $\mathit{Map} \times \mathit{Id} \to \mathit{Int}$ and `update`: $\mathit{Map} \times \mathit{Id} \times \mathit{Int} \to \mathit{Map}$ are part of the domain of maps and have the usual meanings: `lookup` returns the value of an identifier in a map, and `update` modifies the map by adding (or, if it exists, by updating) the binding of an identifier to a value.

In addition to the rules in Figure 3 there are rules induced by the strictness of some statements. For example, the `if` statement is strict only in the first argument, meaning that this argument is evaluated before the `if` statement. This amounts to the following rules (automatically generated by the $\mathbb{K}$ tool):

$$\langle\langle \texttt{if}\ BE\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle BE \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

where BE ranges over $\mathit{BExp}\setminus\{\mathit{false}, \mathit{true}\}$, B ranges over $\{\mathit{false}, \mathit{true}\}$, and $\square$ is a special variable destined to receive the value of BE once it is computed.

3 Formal Background

We assume the reader is familiar with the basics of algebraic specification and rewriting. We also assume a language $\mathcal{L}$ defined by the following ingredients (which will be illustrated below in this section on the IMP language):

1. A many-sorted algebraic signature Σ, which includes at least a sort Cfg for configurations and a subsignature Σ^{Bool} for Booleans with their usual constants and operations. Σ may

$\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle I_1 \texttt{ * } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle I_1 \texttt{ <= } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \mathit{true} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \mathit{false} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{not } \mathit{true} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{not } \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{true} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle S_1 ; S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{\{ } S \texttt{ \}} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{if } \mathit{false} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$
$\quad\langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle \texttt{for } X \texttt{ from } I_1 \texttt{ to } I_2 \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$
$\quad\langle\langle X \texttt{ = } I_1 \texttt{ ;if } X \texttt{ <= } I_2 \texttt{ then\{ } S \texttt{ ;for } X \texttt{ from } I_1 \texttt{ + 1 to } I_2 \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle X \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \texttt{lookup } (\mathit{Env}, I) \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}}$
$\langle\langle X \texttt{ = } I \cdots\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \langle \texttt{update } (\mathit{Env}, X, I)\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}}$

Figure 3: $\mathbb{K}$ Semantics of IMP

also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all data sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{Data}$, and that terms of sort *Cfg* have exactly one subterm denoting statements (which are programs in the syntax of $\mathcal{L}$) remaining to be executed. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(\mathit{Var})$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(\mathit{Var})$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.

2. A Σ-algebra $\mathcal{T}$. Let $\mathcal{T}_s$ denote the elements of $\mathcal{T}$ that have the sort s; the elements of $\mathcal{T}_{Cfg}$ are called *configurations.* $\mathcal{T}$ interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data}-algebra $\mathcal{D}$. $\mathcal{T}$ interprets the non-data sorts as sets of ground terms over the signature

$$(\Sigma \setminus \Sigma^{Data}) \cup \bigcup_{d \in Data} \mathcal{D}_d \tag{1}$$

where $\mathcal{D}_d$ denotes the carrier set of the sort d in the algebra $\mathcal{D}$, and the elements of $\mathcal{D}_d$ are added to the signature $\Sigma \setminus \Sigma^{Data}$ as constants of sort d.

Any *valuation* $\rho : \mathit{Var} \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(\mathit{Var}) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. If $b \in$

$T_{\Sigma,Bool}(Var)$ then we write $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we often write in the sequel $true, false$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}$.

3. A set $\mathcal{S}$ of rewrite rules, whose definition is given later in the section.

We explain these concepts on the IMP example. Nonterminals from the syntax (*Int*, *Bool*, *AExp*, ...) are sorts in Σ. Each production from the syntax defines an operation in Σ; for instance, the production $AExp ::= AExp$ + $AExp$ defines the operation $\_+\_ : AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the configuration sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle X := I \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto 0\; Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{Cfg}(Var)$, where X is a variable of sort *Id*, I is a variable of sort *Int*, C is a variable of sort *Code* (the rest of the computation), and *Env* is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets *Int* as the set of integers, the operations like $+_{Int}$ (cf. Figure 3) as the corresponding usual operation on integers, *Bool* as the set of Boolean values $\{false, true\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $Map_{Id,Int}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers *Id* and I over the integers. The other sorts, *AExp*, *BExp*, *Stmt*, and *Code*, are interpreted in the algebra $\mathcal{T}$ as ground terms over a modification of the form (1) of the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{Int} 0$ `then skip else skip` is intepreted as `if` *true* `then skip else skip` provided $\mathcal{D}_{1 >_{Int} 0} = \mathcal{D}_{true} (= true)$.

We now formally introduce the notions required for defining semantical rules.

Definition 1 (pattern [14]) *A* pattern *is an expression of the form $\pi \wedge b$, where $\pi \in T_{\Sigma,Cfg}(Var)$ are* basic patterns*, $b \in T_{\Sigma,Bool}(Var)$, and $var(b) \subseteq var(\pi)$. If $\gamma \in \mathcal{T}_{Cfg}$ and $\rho \colon Var \to \mathcal{T}$ we write $(\gamma, \rho) \models \pi \wedge b$ for $\gamma = \rho(\pi)$ and $\rho \models b$.*

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy. In [14] patterns are encoded as FOL formulas, hence the conjunction notation $\pi \wedge b$. In this paper we keep the notation but separate basic patterns from constraining formulas. We identify basic patterns π with paterns $\pi \wedge true$. Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0$.

Definition 2 (semantical rule and transition system) *A* rule *is a pair of patterns of the form $l \wedge b \Rightarrow r$ (note that r is the pattern $r \wedge true$). Any set $\mathcal{S}$ of rules defines a labelled transition system $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ such that $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ iff there are $(l \wedge b \Rightarrow r) \in \mathcal{S}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$.*

A configuration γ is *final* if its program subterm is empty. A configuration γ is a *deadlock* if it is not final and there is no configuration γ' such that $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$. Deadlocks are erroneous program terminations, e.g., division-by-zero attempts. A language is *deterministic* if its transition system $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ is deterministic.

Assumption 1 *We assume that the ransition system $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ is deterministic.*

We shall be using unification in our program-equivalence deductive system. We call *symbolic unifier* of two terms t_1, t_2 any substitution $\sigma : var(t_1) \uplus var(t_2) \to T_{\Sigma}(Z)$ for some set Z of variables such that $t_1\sigma = t_2\sigma$. We call a *concrete unifier* of terms t_1, t_2 any valuation $\rho : var(t_1) \uplus var(t_2) \to \mathcal{T}$ such that $t_1\rho = t_2\rho$.

Assumption 2 *For all rules $(l \wedge b \Rightarrow r) \in \mathcal{S}$ and all patterns $\pi \in T_{\Sigma,Cfg}(Var)$ with $var(l) \cap var(\pi) = \emptyset$, there is a finite, possibly empty set $U(\pi, l)$ of symbolic unifiers of π and l, which satisfy the property that for all concrete unifiers ρ of π and l, there exist substitutions $\sigma \in U(\pi, l)$ and valuations η such that $\sigma\eta = \rho$.*

In related work [15] we prove that the above assumption can always be satisfied, by implementing unification with the rules of $\mathcal{L}$ by the *matching* with the rules of a language $\mathcal{L}^{sym}$, which extends the syntax of and semantics of $\mathcal{L}$, such that the programs in $\mathcal{L}^{sym}$ are exactly the symbolic programs of $\mathcal{L}$ (and the symbolic execution of programs in $\mathcal{L}$ is the usual execution of programs in $\mathcal{L}^{sym}$). We illustrate how this is done via an example; other examples follow in the paper.

Example: Consider the pattern $\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ of sort *Cfg*, where B is a variable of sort *Bool* and S_1, S_2 are variables of sort *Stmt*, and consider also the rule $\langle\langle(\texttt{if } \mathit{true} \texttt{ then } S'_1 \texttt{ else } S'_2) \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S'_1 \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. Here we have filled in the "..." from Figure 3 with actual variables, and the rule's variable were chosen so that they are distinct from those in the formula. Let π denote the pattern and l the left-hand side of the rule. The set $U(\pi, l)$ is a singleton given by the substitution $\sigma = (B \mapsto \mathit{true}, S'_1 \mapsto S_1, S'_2 \mapsto S_2, M' \mapsto M)$. On the other hand, l does not match π because the constant leaf *true* of l does not match the variable B in π. However, the rule can be equivalently rewritten

$$\langle\langle(\texttt{if } B' \texttt{ then } S'_1 \texttt{ else } S'_2) \curvearrowright\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B' = \mathit{true} \Rightarrow \langle\langle S'_1 \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

and now, there is match between the configuration l' from the left-hand side of the new rule and π, i.e., $(B' \mapsto B, S'_1 \mapsto S_1, S'_2 \mapsto S_2, M' \mapsto M)$. This match, combined with the condition $B' = \mathit{true}$, amount to the above symbolic unifier σ.

4 Defining Program Equivalence

We define in this section our notion of program equivalence. We base our definition on the transition system $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$, whose states $\mathcal{T}_{Cfg}$ are configurations, and $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$ is the transition relation defined in the previous section (Definition 2). Our goal is to have a definition of equivalence that is equally suitable for terminating programs and non-terminating ones and for symbolic and concrete ones.

A natural approach (already chosen by [1]) is to use *strong bisimulation*: a symmetrical relation $R \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is a strong bisimulation if for all $(\gamma_1, \gamma_2) \in R$, when $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_1$, there is a transition $\gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_2$ such that $(\gamma'_1, \gamma'_2) \in R$. However, for our purpose such relations are too strong; e.g., the assignment i = 2 is not equivalent to the sequence i = 1; i = 2 because, starting from $i = 0$, the former reaches $i = 2$ in one semantical step, whereas the latter cannot.

Hence, we need to alter strong bisimulation for our purposes. We do it, first, by removing the constraint that each step of one program is matched by exactly one step of the other one, and second, by requiring that our relation be upper bounded by a certain relation $O \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ called the *observation relation.*

Definition 3 (O-weak bisimulation) *An O-*weak bisimulation *is a relation $R \subseteq O$ satisfying: for all $(\gamma_1, \gamma_2) \in R$,*

- *if $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_1$ then $\gamma'_1 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma''_1$ and $\gamma_2 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma''_2$, for some $(\gamma''_1, \gamma''_2) \in R$*
- *if $\gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_2$ then $\gamma_1 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma''_1$ and $\gamma'_2 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma''_2$ for some $(\gamma''_1, \gamma''_2) \in R$.*

In the sequel we assume O to be an arbitrary, fixed parameter to our definitions. We omit it and only write "weak bisimulation" instead of "O-weak bisimulation". We now have our definition of program (actually, of configuration) equivalence:

Definition 4 (Configuration Equivalence) *Configurations γ_1, γ_2 are* equivalent*, written $\gamma_1 \sim \gamma_2$, if there is a weak bisimulation R such that $(\gamma_1, \gamma_2) \in R$.*

Example:
The configurations $\gamma_1 \triangleq \langle\langle \texttt{x = 2}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\gamma_1' \triangleq \langle\langle \texttt{x = 1; x = x+1}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ are equivalent when O is defined by the value of x be equal in both γ_1, γ_2. The "witness" weak bisimulation R for the equivalence $\gamma_1 \sim \gamma_1'$ is defined by $\{(\gamma_1, \gamma_1'), (\gamma_2, \gamma_2)\}$, where $\gamma_2 \triangleq \langle\langle \cdot \rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. The relation O gives us quite a lot of expressiveness for capturing various kinds of program equivalences. For example, *partial* equivalence [2] is: two programs are equivalent if, whenever presented with the same input, if they both terminate they produce the same output. This can be encoded by including cells in the configuration for the input and output, and by including in O the pairs of configurations satisfying: if their programs are both empty and their inputs are equal then their outputs are equal. Also, *full* equivalence from [2] is: two programs are equivalent if, whenever presented with the same input, they either both terminate and produce the same output, or they both do not terminate. This is captured by adding to the above relation all pairs of configurations from which there is an infinite execution starting from both configurations of the pair.

5 A Logic for Program Equivalence

We present in this section a logic for program equivalence. We first present the logic's syntax, then its semantics, and finally the notion of validity for formulas.

Definition 5 (Formulas) *A formula is an expression of the form $\pi_1 \sim \pi_2$ if C where $\pi_1, \pi_2 \in T_{\Sigma,Cfg}(Var)$ are patterns and $C \in T_{\Sigma,Bool}(Var)$.*

Example: Assume that the signature Σ for the language IMP contains a predicate `occurs` : *Id* $\times$ *Stmt* $\to$ *Bool* expressing the fact that an identifier occurs in a statement. A formula expressing the equivalence of the programs in Example 1 is

$$\begin{array}{l} \langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}S\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim \\ \langle\langle I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\texttt{+1\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \ \mathit{if}\ \ \texttt{not occurs(}\ I, S\ \texttt{)} \end{array}$$

where M a variable of sort *Map*. The condition says that the loop counter I does not occur in the loop body S. It is essential for the formula's validity.

We now define two semantics for formulas $f \triangleq \pi_1 \sim \pi_2$ *if* C. The first one, denoted by $(\!|f|\!)$, is the set of pairs of configurations γ_1, γ_2 that satisfy, respectively, the patterns $\pi_1 \wedge C$ and $\pi_2 \wedge C$ by means of one valuation (the same valuation for both γ_1, γ_2). The second one, denoted by $[\![f]\!]$, excludes from $(\!|f|\!)$ the pairs of configurations from which at least one component eventually leads to a deadlock.

Definition 6 (Semantics) $(\!|f|\!) \triangleq \{(\gamma_1, \gamma_2) \mid \exists \rho : Var \to \mathcal{T}.(\gamma_i, \rho) \models \pi_i \wedge C, i = 1, 2\}$*, and* $[\![f]\!] \triangleq \{(\gamma_1, \gamma_2) \in (\!|f|\!) \mid .\forall i \in \{1, 2\} \forall \gamma \in \mathcal{T}_{Cfg}.\, \gamma_i {\Rightarrow^*}_{\mathcal{S}}^{\mathcal{T}} \gamma$ *implies γ is no deadlock*$\}$.

We now define what it means for a formula f to be *valid*. Intuitively, we want to capture the idea that all configurations pairs $(\gamma_1, \gamma_2) \in [\![f]\!]$ satisfy $\gamma_1 \sim \gamma_2$ according to Definition 4. We use the $[\![\cdot]\!]$ semantics (not the $(\!|\cdot|\!)$ one) because we are not interested in deadlocks. This is not really a restriction since deadlocks can be turned into final configurations by adding rules and, e.g., setting the content of some cell, say, `error`, to some value encoding the deadlock situation.

Definition 7 (Validity) *A formula f is valid, written $\mathcal{S} \models f$, if $[\![f]\!] \neq \emptyset$ whenever $(\!|f|\!) \neq \emptyset$, and for all $\gamma_1, \gamma_2 \in [\![f]\!]$, $\gamma_1 \sim \gamma_2$.*

6 Auxiliary Operations: Derivatives and Conjunction

Our proof system consists in symbolically executing formulas according to the semantics of the language $\mathcal{L}$. This is achieved using the notion of *derivative.*

Definition 8 (Derivatives) *Given a formula $g \triangleq \pi_1 \sim \pi_2$ if C, its derivatives are the formulas in the set $\Delta(g) = \Delta^l(g) \cup \Delta^r(g)$, where $\Delta^l(g), \Delta^r(g)$ are the smallest sets defined by: for each $(l \wedge C' \Rightarrow r) \in \mathcal{S}$, $\sigma^l \in U(\pi_1, l)$, $\sigma^r \in U(\pi_2, r)$:*

- $(r\sigma^l \sim \pi_2 \text{ } \mathit{if} \text{ } (C \wedge C')\sigma^l \wedge \bigwedge\sigma^l) \in \Delta^l(g)$,
- $(\pi_1 \sim r\sigma^r \text{ } \mathit{if} \text{ } (C \wedge C')\sigma^r \wedge \bigwedge\sigma^r) \in \Delta^r(g)$

where $\bigwedge\sigma \triangleq \bigwedge_{x \in dom(\sigma)}(x = \sigma(x))$, and $dom(\sigma)$ denotes the subset of the gobal set Var of variables where the substitution σ is not the identity. We naturally extend derivatives to sets F of formulas by $\Delta(F) = \bigcup_{f \in F} \Delta(f)$.

Remark 1 *In Definition 8 we assume $var(l) \cap var(g) = \emptyset$, which can always be obtained by renaming the variables in the rewrite rule.*

Example: Let B be a variable of sort *Bool* and S_1, S_2 be variables of sort *Stmt*. We consider the formula f below and compute its left-derivatives:

$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if } B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$
$$\mathit{if}\ B' = \neg B$$

The rules with a nonempty set of unifiers with the patterns in the formula are

$$\langle\langle(\texttt{if } \mathit{true} \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$
$$\langle\langle(\texttt{if } \mathit{false} \texttt{ then } S_1' \texttt{ else } S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

The formula f has two left-derivatives, i.e., $\Delta^l(f)$ are the formulas in the set

$$\langle\langle S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if}\, B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \mathit{if}\ B' {=} \neg B \wedge B {=} \mathit{true}$$
$$\langle\langle S_2\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle \texttt{if}\, B' \texttt{ then } S_2 \texttt{ else } S_1\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \mathit{if}\ B' {=} \neg B \wedge B {=} \mathit{false}$$

where $B = \mathit{true}$ and $B = \mathit{false}$ are induced by the symbolic unifiers: $B \mapsto \mathit{true}$, $S_1' \mapsto S_1$, $S_2' \mapsto S_2$, $M' \mapsto M$ and, respectively, $B \mapsto \mathit{false}$, $S_1' \mapsto S_1$, $S_2' \mapsto S_2$, $M' \mapsto M$. The superfluous equalities $S_1' = S_1$, $S_2' = S_2$, $M' = M$ were removed from conditions since S_1', S_2', and M' do not occur in the rest of the formula.

Another auxiliary operation used in our proof system is *conjunction* of formulas. We need it in order to compute the subsets of configuration pairs, denoted by formulas, which are included in the observation relation O (cf. Section 4).

Definition 9 *For formulas $f : \pi_1 \sim \pi_2$ if C and $g : \pi_1' \sim \pi_2'$ if C', let $f \wedge g =$ $\{\pi_1\sigma_1 {\sim} \pi_2\sigma_2 \text{ } \mathit{if} \text{ } (C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge\sigma_1 \wedge \bigwedge\sigma_2 \mid \sigma_1 \in U(\pi_1, \pi_1'),\ \sigma_2 \in U(\pi_2, \pi_2')\}$.*

Example: Let f be the formula in Example 6 and let g denote the formula $\langle\langle P_1\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle M''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *if* $M' = M''$. We denote by π_1, π_1' and π_2, π_2' their left and right-hand sides, respectively. Then, $U(\pi_1, \pi_1')$ can be computed by matching, and consists of the unique substitution $\sigma_1 = (P_1 \mapsto \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, M' \mapsto M)$. Similarly, $U(\pi_2, \pi_2')$ consists of the substitution $\sigma_2 = (P_2 \mapsto \texttt{if } B' \texttt{ then } S_2 \texttt{ else } S_1, M'' \mapsto M)$. Thus, if we remove the conditions $M' = M''$, $\bigwedge\sigma_1$, and $\bigwedge\sigma_2$ (which are superfluous here since they constrain variables not occuring in the rest of the result), $f \wedge g$ is syntactically equal to f. This is consistent with the fact that $\wedge$ is, semantically speaking, intersection, because we have $(\!|f|\!) \subseteq (\!|g|\!)$ and thus $(\!|f \wedge g|\!) = (\!|f|\!) \cap (\!|g|\!) = (\!|f|\!)$.

7 A Circular Proof System

In this section we define a three-rule proof system for proving program equivalence. It is inspired from *circular coinduction* [12], a coinductive proof technique for infinite data structures and coalgebras of expressions [16].

Remember that we have fixed an observation relation O. We assume a set of formulas Ω such that $(\!|\Omega|\!) = O$. We also assume that for all $h \in \Omega$ and for all formula f, the conjunction $f \wedge h$ can be computed according to Definition 9:

Assumption 3 *For all $(\pi_1' \sim \pi_2'$ if $C) \in \Omega$ and all $\pi \in T_{\Sigma,Cfg}(Var)$ with $(var(\pi_1) \cup var(\pi_2)) \cap var(\pi) = \emptyset$, there are two finite, possibly empty sets $U(\pi, \pi_1')$ and $U(\pi, \pi_1')$ of symbolic unifiers of π, π_1' and of π, π_2', respectively.*

Let also $\vdash$ be an entailment relation satisfying $\mathcal{S}, F \vdash g$ implies $(\mathcal{S} \models g$ or $(\!|g|\!) \subseteq (\!|F|\!))$. The set Ω and the relation $\vdash$ are parameters of our proof system:

Definition 10 (Circular Proof System)

$$\textit{[Axiom]}\ \frac{}{\mathcal{S}, F \vdash^{\circlearrowleft} \emptyset}$$

$$\textit{[Reduce]}\ \frac{\mathcal{S}, F \vdash g \qquad \mathcal{S}, F \vdash^{\circlearrowleft} G}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{g\}}$$

$$\textit{[Derive]}\ \frac{\mathcal{S}, F \cup F' \vdash^{\circlearrowleft} G \cup \Delta(g) \qquad \mathcal{S}, g \wedge \Omega \vdash F'}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{g\}}\ \textit{if}\ \Delta(g) \neq \emptyset$$

where $g \wedge \Omega$ denotes the set $\{g \wedge h \mid h \in \Omega\}$.

[Axiom] says that when an empty set of goals is reached, the proof is finished.

The [Reduce] rule says that if a given goal g from the current set of goals $G \cup \{g\}$ is discharged by the entailment $\vdash$ then it is eliminated from the goals.

The last rule, [Derive], is the most complex, and uses the auxiliary constructions (derivative, conjunction) introduced earlier. It says that any given goal g from the current set of goals, with a nonempty set $\Delta(g)$ of derivatives, can be replaced in the goals to be proved with the set $\Delta(g)$; and, simultaneously, any set of formulas F' that can be $\vdash$-entailed from $\mathcal{S}, g \wedge \Omega$ can be added as hypotheses. Note that the application of the [Derive] rule is nondeterministic in the choice of hypotheses F', which depend on the parameters $\vdash$ and Ω of the proof system.

Theorem 1 (soundness of $\vdash^{\circlearrowleft}$) *Let Γ be a set of formulas such that $(\!|\Gamma|\!) \subseteq (\!|\Omega|\!)$ and for all $g \in \Gamma$, $[\![g]\!] \neq \emptyset$. If $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ then $\mathcal{S} \models \Gamma$.*

Note that we require $(\!|\Gamma|\!) \subseteq (\!|\Omega|\!)$ because otherwise the goals Γ have no chance of being valid. The asumption for all $g \in \Gamma$, $[\![g]\!] \neq \emptyset$ (that implies $(\!|g|\!) \neq \emptyset$) is made for ensuring that g is not *vacuously* valid. Note also that initially, the set of hypotheses, denoted by F in the proof system, is empty: $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ is actually an abbreviation for the full notation $\mathcal{S}, \emptyset \vdash^{\circlearrowleft} \Gamma$.

We now show that the circular proof system, when it terminates, always provides an answer (positive or negative) to the question $\mathcal{S} \models \Gamma$. Thus, in addition to soundness we have a *weak completeness* result. The result is "weak" because it assumes termination of the proof system. It ensures that we have a decision procedure for the equivalence of concrete, terminating programs.

In order to achieve weak completeness we need the following adaptations of Definition 8: we only keep the formulas with a *satisfiable condition*, i.e., we eliminate "empty" formulas f with $(\!|f|\!) = \emptyset$. We also need additional assumptions. The first one says that non-derivable goals g

that denote observationally equivalent configuration pairs are valid, and are discharged by the entailment $\vdash$. The second one says that deadlocks are not observationally equivalent to anything.

Assumption 4 *For all formulas g such that $(\!|g|\!) \subseteq (\!|\Omega|\!)$ and $\Delta(g) = \emptyset$, $\mathcal{S} \vdash g$; and for all configurations γ_1, γ_2, if γ_1 or γ_2 are deadlocks then $(\gamma_1, \gamma_2) \notin (\!|\Omega|\!)$.*

Theorem 2 (Weak Completeness of $\vdash^{\circlearrowleft}$) *Assume $\mathcal{S} \models \Gamma$ and the proof system $\vdash^{\circlearrowleft}$ terminates on Γ. Then, $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$.*

Given a set of goals Γ, the proof system $\vdash^{\circlearrowleft}$ may *terminate successfully* on it, which means if generates a tree that has at least one "empty" leaf (generated by [Axiom]). The proof system may also *terminate unsucessfully* when it generates a finite tree and cannot expand it (i.e., it is blocked) and moreover that tree does not have any empty leaf. The proof system terminates on Γ if it terminates either sucessfully or unsucessfully. Weak completeness thus says that if a set of goals is valid and the proof system terminates on it, then it terminates successfully.

Together, the soundness and weak completeness say that, if the proof system applied to a given set of goals terminates, then termination is successful if and only if the set of goals is valid. That is, when it terminates, the proof system correctly solves the program-equivalence problem. Of course, termination cannot be guaranteed, because the equivalence problem is undecidable. It does terminate on goals in which both programs terminate (because eventually the set of derivatives becomes empty) and also for goals in which one or both of the programs does not terminate, provided they behave in a certain regular way. *Example:* We show the application of our proof system for proving the equivalence of our *for* and *while* programs formalised as the validity of the formula f (in which we assume for simplicity that the symbolic statement S is terminating; non-terminating statements can be handled as well but complicate the example):

$$\begin{aligned}&\langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}\,S\,\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim\\ &\langle\langle I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}\,S\ \texttt{;}\ I\ \texttt{=}\ I\texttt{+1\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\\ &\textit{if}\ \ \texttt{not occurs(}\ I, S\ \texttt{)}\end{aligned} \tag{2}$$

when the observation relation is denoted by the set $\Omega = \{\langle\langle P_1\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle M''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\ \textit{if}\, M' = M''\}$. The observation relation says that two configurations are observationally equivalent whenever they have equal environments.

The first applied rule is [Derive], which adds to the initially empty set of hypotheses the formula f, simulataneously replacing it in the goals with $\Delta(f)$. (f can be added to the hypoteses because $(\!|f|\!) \subseteq (\!|\Omega|\!)$, which implies $\Omega \wedge f \vdash f$).

After a certain number of applications of the [Derive] rule, the set of goals becomes (after some simplifications, which consist in removing goals with unsatisfiable conditions and logically simplifying the conditions of the remaining goals; note that A and B became (symbolic) values due to the `strict` attribute):

$$\langle\langle\rangle_{\mathsf{k}}, \langle \texttt{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \ \sim\ \langle\langle\rangle_{\mathsf{k}}, \langle \texttt{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\ \textit{if}\, A >_{\mathit{Int}} B \tag{3}$$

$$\begin{aligned}&\langle\langle \texttt{for}\ I\ \texttt{from}\ A +_{\mathit{Int}} 1\ \texttt{to}\ B\ \texttt{do\{}\,S\,\texttt{\}}\rangle_{\mathsf{k}}, \langle \texttt{followup}(S, \texttt{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim\\ &\langle\langle I\texttt{=}A +_{\mathit{Int}} 1\texttt{;while}\,I\texttt{<=}B\,\texttt{do\{}S\texttt{;}I\texttt{=}I\texttt{+1\}}\rangle_{\mathsf{k}}, \langle \texttt{followup}(S, \texttt{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\\ &\textit{if}\ \ \texttt{not occurs(}\ I, S\ \texttt{)} \wedge A \leq_{\mathit{Int}} B\end{aligned} \tag{4}$$

where `followup` (S, M) denotes the effect of executing statement S on map M. (Remember that S is terminating, so `followup` (S, M) is defined. Moreover, for each concrete statement P that is an instance of S we have the implicit definition `followup` $(P, M) = M'$ iff $\langle\langle P\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \langle\langle\rangle_{\mathsf{k}}, \langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$.)

The first goal is discharged by the [Reduce] rule (based on the fact that the $\vdash$ relation "knows" that goals with same left and right-hade side are valid). The second goal f' is actually an *instance* of the first one: i.e., $(\!|f'|\!) \subseteq (\!|f|\!)$ since any concrete instance of f' is also a concrete instance of f. Thus, $\mathcal{S}, f \vdash f'$, and since f was added to the set hypotheses by the first application of [Derive], f' is eliminated by the [Reduce] rule. The set of goals to be proved is now empty; the proof system has terminated successfully, meaning that the formula f is valid.

8 A Prototype Implementation

$\mathbb{K}$ [13] is a framework for defining the formal operational semantics of programming languages. One component of the framework is a compiler of $\mathbb{K}$ definitions to Maude [17] specifications, which can then be used for executing programs and for analysing them. $\mathbb{K}$ also offers some support for symbolic calculus [18] (in progress), including a connection to the Z3 SMT solver [19]. We have used these components in a prototype tool implementing our deductive system for program equivalence. Here we describe how the proof system proposed in this paper is implemented for the IMP language. This description is generic enough so that one can be thought as being a methodology that can be applied to any other language defined in $\mathbb{K}$.

The first step is extending the definition of the language with the symbolic calculus support following the methodology described in [18]. This adds, among other features, a new cell `cond` that stores the path condition of the current execution path. We use this cell for storing the conditions of goals. Then, we:

1. extend the definition of the language with the syntax of equivalence formulas and rules for building the initial configuration;
2. extend the configuration for storing formulas as pairs of original configurations and using the cell `cond` for storing the condition;
3. add a new sub-configuration for storing the circular hypotheses F;
4. add rules giving semantics to the symbolic statements in a conservative way;
5. add rules storing new circular hypotheses in the configuration;
6. add syntax and rules that define the entailment relation between circular hypotheses F and the current formula;
7. add syntax and rules that define the basic entailment and the observation relation. In the prototype the observation relation is specified by a set of variables required to have the same values in the two configurations of the formula, and the basic entailment discharges a goal when its configurations are in the observation relation and their `k` cells have the same content;
8. define heuristics to decide the following facts about the current formula: whether it belongs to the observation relation, or it is a consequence of the circular hypotheses, or it must be added to the circular hypotheses.

One may see all the above as being the definition of a new language, where the programs represents the equivalence formulas, and the successful execution of a formula proves its validity. The steps 6 and 8 are the most difficult to implement. We briefly describe their implementation in the prototype.

An equivalence formula $\pi_1 \sim \pi_2$ *if* C for IMP can be shortly written as $\langle p_1\rangle_{\mathsf{k}}\langle\rho_1\rangle_{\mathsf{env}}\langle p_2\rangle_{\mathsf{k}}\langle\rho_2\rangle_{\mathsf{env}}\langle C\rangle_{\mathsf{cond}}$, where the pattern π_i is given by the contents of the corresponding `k` and `env` cells. A *program substitution* between the programs p and p' is a mapping σ : *program-variables*$(p) \cup$ *symbolic-expressions*$(p) \to$ *program-variables*$(p') \cup$ *symbolic-expressions*(p') such that $\sigma(p) = p'$.

Recall that the current goal g is a consequence of the circular hypotheses F iff $(\!|g|\!) \subseteq (\!|F|\!)$. We define $F \vdash g$ iff there is a program substitution σ and $f \triangleq \langle p'\rangle_{\mathsf{k}}\langle\rho'\rangle_{\mathsf{env}}\langle C'\rangle_{\mathsf{cond}}$ in F such that $\sigma(p') = p$, $C \implies C'$ (this is checked by calling a SMT solver), and ρ and ρ' are in observational relation. Then we have $F \vdash g$ implies $(\!|g|\!) \subseteq (\!|F|\!)$. This is the implementation of step 6.

The step 8 is implemeted using labeled statements: each time two statements with the same label are on the top of the `k` cell storing the formula, a set of semantic rules decide which one of the three cases (described in the step 8) holds, and take the corresponding action. The implementation is available online at `http://fmse.info.uaic.ro/tools/K/?tree=examples/prog-equiv/peq.k`.

9 Conclusion and Future Work

We have presented a definition for program equivalence, a logic that encodes this definition in its formulas, and a proof system for the logic, which is proved sound and weakly complete. A prototype implementation for the proof system in the $\mathbb{K}$ framework was also presented and illustrated on a simple but paradigmatic example of equivalent programs in a language also defined in the $\mathbb{K}$ framework. The proposed approach is general: it does not depend on $\mathbb{K}$, but only requires a formal semantics of the language of interest presented as a term-rewriting system. The chosen equivalence relation is a weak bisimulation, which is parametric in a certain observation relation. We show the approach is applicable for concrete and symbolic program, as well as for terminating and non-terminating ones.

Future Work We are planning to apply our deductive system for proving the correctness of a compiler between two languages (as part of another project we are involved in). The source language is a stack-based language with control structures (loops, conditionals, function calls). The target is also stack-based but only has (possibly, conditional) jumps. The correctness of the compiler amounts to proving the equivalence of several pairs of symbolic programs; in each pair, one component denotes a source-language control structure, and the other component is the translation of that control structure in the target language using jumps. Our ability to check equivalence of *symbolic* programs is essential for this.

References

[1] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 327–337. ACM, 2009.

[2] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, 2008.

[3] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 2012. To appear, available online first at `http://dx.doi.org/10.1002/stvr.1472`.

[4] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2012.

[5] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[6] George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *PLDI*, pages 83–94. ACM, 2000.

[7] Andrew M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 378–412, London, UK, UK, 2002. Springer-Verlag.

[8] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2005.

[9] Sorin Craciunescu. Proving the equivalence of clp programs. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2002.

[10] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717. Springer, 2012.

[11] Fabio Somenzi and Andreas Kuehlmann. *Electronic Design Automation For Integrated Circuits Handbook*, volume 2, chapter 4: Equivalence Checking. Taylor & Francis, 2006.

[12] Grigore Roşu and Dorel Lucanu. Circular coinduction – a proof theoretical foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.

[13] G. Roşu and T.-F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[14] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012. To appear.

[15] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A Generic Approach to Symbolic Execution. Rapport de recherche RR-8189, INRIA, December 2012.

[16] M. Bonsangue, G. Caltais, E. Goriac, D. Lucanu, Jan J. M. M. Rutten, and A. Silva. A decision procedure for bisimilarity of generalized regular expressions. In *Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF 2010)*, volume 6527 of *LNCS*, pages 226–241. Springer, 2011. An extended version will appear in Science of Programming.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[18] A. Arusoaie, D. Lucanu, and V. Rusu. A generic approach to symbolic execution. Technical Report RR-8189, INRIA, December 2012.

[19] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

Appendix: Proofs

Lemma 1 *$(\!|f|\!) \subseteq (\!|g|\!)$ implies $[\![f]\!] \subseteq [\![g]\!]$.*

Proof: Since $[\![f]\!] \subseteq (\!|f|\!)$ from the hypothesis we get $[\![f]\!] \subseteq (\!|g|\!)$. For all $(\gamma_1, \gamma_2) \in [\![f]\!]$, there are infinite computations starting in γ_1 and γ_2, hence, $(\gamma_1, \gamma_2) \in [\![g]\!]$. $\square$

Notation. If $\rho : Var \to \mathcal{T}$ and $V, V' \subseteq Var$ s.t. $V \cap V' = \emptyset$, then $\rho|_V$ denotes the function $\rho|_V : V \to \mathcal{T}$ given by $x\rho|_V = x\rho$ for all $x \in V$ (the restriction of ρ to V) and $\rho|_V \cup \rho|_{V'} = \rho|_{V \cup V'}$. Obviously, $\rho = \rho_V \cup \rho_{\overline{V}}$, where $\overline{V}$ denotes the complement of V.

If $\sigma : Var \to T_\Sigma(Var)$ then $dom(\sigma) = \{x \in Var \mid x\sigma \neq x\}$ and $ran(\sigma) = \bigcup_{x \in dom(\sigma)} var(x\sigma)$. If $dom(\sigma) \cap dom(\sigma') = \emptyset$, then $\sigma \cup \sigma'$ is the substitution given by

$$x(\sigma \cup \sigma') = \begin{cases} x\sigma & x \in dom(\sigma) \\ x\sigma' & x \in dom(\sigma') \\ x & \text{otherwise} \end{cases}$$

Remark 2 *In the following results we assume that for each formula $f \triangleq \pi_1 \sim \pi_2$ if C we have $var(\pi_1) \cap var(\pi_2) = \emptyset$. This is not a restriction because for each shared variable x we create a fresh variable x', replace e.g. in π_2 x with x', and take $C \wedge (x = x')$ as the new condition of the formula.*

Lemma 2 *Let $g \triangleq \pi_1 \sim \pi_2$ if C be a formula. For all $(\gamma_1, \gamma_2) \in (\!|g|\!)$, if $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_1$ then there exists $g' \in \Delta^l(g)$ such that $(\gamma'_1, \gamma_2) \in (\!|g'|\!)$.*

Proof: Since $(\gamma_1, \gamma_2) \in (\!|g|\!)$, there exists $\rho : Var \to \mathcal{T}$ such that $(\gamma_i, \rho) \models \pi_i \wedge C$ for $i = 1, 2$. Since $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_1$, by the definition of $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$ there exists $(l \wedge b \Rightarrow r) \in R$ and $\rho' : Var \to \mathcal{T}$ such that $(\gamma_1, \rho') \models l \wedge b$ and $\gamma'_1 = r\rho'$. Recall that $var(l) \cap (var(\pi_1) \cup var(\pi_2)) = \emptyset$, by Remark 1. We consider the valuation $\rho'' = \rho|_{\overline{var(l)}} \cup \rho'|_{var(l)}$; we have $\pi_1\rho'' = l\rho'' = \gamma_1$, $\rho'' \models (C \wedge b)$, and by Assumption 2 there exists $\sigma \in U(\pi_1, l)$ and η such that $l\sigma = \pi_1\sigma$ and $\sigma\eta = \rho''$.

Moreover $\eta : Var \to \mathcal{T}$ can be chosen so that $x\eta = x\rho'$ for all $x \in var(l)$ (we assume $var(l) \cap ran(\sigma) = \emptyset$, which can always hold by renaming variables in $ran(\sigma)$) and $x\eta = x\rho$ for all $x \in var(\pi_1)$ (since we assume $var(\pi_1) \cap ran(\sigma) = \emptyset$)

Definition 8 ensures that the formula $g' \triangleq (r\sigma \sim \pi_2$ *if* $(C \wedge b)\sigma \wedge \bigwedge\sigma) \in \Delta^l(g)$. Since:

$$\begin{aligned}
\gamma'_1 &= r\rho' && \text{(by the definition of } \gamma'_1\text{)} \\
&= r(\rho'|_{\overline{var(l)}} \cup \rho'|_{var(l)}) \\
&= r(\rho|_{\overline{var(l)}} \cup \rho'|_{var(l)}) && (\rho' \text{ chosen s.t. } \rho'|_{\overline{var(l)}} = \rho|_{\overline{var(l)}} \text{ allowed by Rem. 1)} \\
&= r\sigma\eta && \text{(by the def. of } \rho'', \sigma \text{ and } \eta\text{)} \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \sigma\eta|_{var(\pi_2)}) \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) && (\sigma\eta|_{var(\pi_2)} = \rho|_{var(\pi_2)} \text{ by def. of } \rho'', \sigma \text{ and } \eta) \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \sigma\rho|_{var(\pi_2)}) && (\sigma \text{ is the identity on } var(\pi_2)) \\
&= r\sigma(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) \\
&= (r\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) && (5) \\
\gamma_2 &= \pi_2\rho && \text{(by the definition of } \gamma_2\text{)} \\
&= \pi_2(\rho|_{var(\pi_2)} \cup \rho|_{\overline{var(\pi_2)}}) \\
&= \pi_2(\rho|_{var(\pi_2)} \cup \eta|_{\overline{var(\pi_2)}}) && \text{(since valuations on } \overline{var(\pi_2)} \text{ do not affect } \pi_2\text{)}
\end{aligned}$$

$$= \pi_2(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) \tag{6}$$

and

$$\begin{aligned}((C \wedge b)\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) &= (C \wedge b)(\sigma\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)})\\ &= (C \wedge b)\sigma\eta\\ &= (C \wedge b)\rho''\\ &= \mathit{true}\end{aligned} \tag{7}$$

and

$$\begin{aligned}&(\bigwedge\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)})\\ &= \bigwedge_{x\in var(l)\cup var(\pi_1)} (x = x\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)})\\ &= \bigwedge_{x\in var(l)\cup var(\pi_1)} (x = x\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \eta|_{var(\pi_2)}) \qquad (dom(\sigma) \cap var(\pi_2) = \emptyset \text{ by Rem. 1,2})\\ &= \bigwedge_{x\in var(l)\cup var(\pi_1)} (x = x\sigma)\eta\\ &= \left(\bigwedge_{x\in var(l)} (x = x\sigma)\eta\right) \wedge \left(\bigwedge_{y\in var(\pi_1)} (y = y\sigma)\eta\right)\\ &= \left(\bigwedge_{x\in var(l)} x\rho' = x\sigma\eta\right) \wedge \left(\bigwedge_{y\in var(\pi_1)} y\rho = y\sigma\eta\right) \qquad (x\eta = x\rho', y\eta = y\rho)\\ &= \left(\bigwedge_{x\in var(l)} x\rho'' = x\rho''\right) \wedge \left(\bigwedge_{y\in var(\pi_1)} y\rho'' = y\rho''\right) \qquad (\text{def. of. } \rho'')\\ &= \mathit{true}\end{aligned} \tag{8}$$

it follows that (5)-(8) imply $(\gamma_1', \gamma_2) \in (\!|g'|\!)$ using the valuation $\eta \cup \rho|_{var(\pi_2)}$. $\Box$

Corollary 1 *For all $(\gamma_1, \gamma_2) \in [\![g]\!]$, if $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$ then there exists $g' \in \Delta^l(g)$ such that $(\gamma_1', \gamma_2) \in [\![g']\!]$.*

Proof: Let $(\gamma_1, \gamma_2) \in [\![g]\!]$ and $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$. Since $[\![g]\!] \subseteq (\!|g|\!)$, by Lemma 2 there exists $g' \in \Delta^l(g)$ such that $(\gamma_1', \gamma_2) \in (\!|g'|\!)$. Since $(\gamma_1, \gamma_2) \in [\![g]\!]$, neither γ_1, nor γ_2 may lead to deadlocks, and, by determinism, neither does γ_1'. Thus, $(\gamma_1', \gamma_2) \in [\![g']\!]$. $\Box$

Lemma 3 *Let $f \triangleq \pi_1 \sim \pi_2$ if C be a formula. For all $(\gamma_1, \gamma_2) \in [\![f]\!]$:*

- *if $\Delta^l(f) \neq \emptyset$ then there is $f' \in \Delta^l(g)$ and $(\gamma_1', \gamma_2) \in [\![f']\!]$ such that $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$;*
- *if $\Delta^r(f) \neq \emptyset$ then there is $f' \in \Delta^r(g)$ and $(\gamma_1, \gamma_2') \in [\![f']\!]$ such that $\gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2'$.*

Proof: We prove the first statement; the second one is proved similarly. Assume that γ_1 is final. Then, the pattern π_1 contains an empty program. This contradicts $\Delta^l(f) \neq \emptyset$. Since $(\gamma_1, \gamma_2) \in [\![f]\!]$, by definition of $[\![\cdot]\!]$, γ_1 is not a deadlock. Since γ_1 is neither final nor a deadlock, using the determinism of $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$, there exists exactly one transition $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$. and Corollary 1 concludes the proof. $\Box$

Lemma 4 *Let $f \triangleq \pi_1 \sim \pi_2$ if C and $g \triangleq \pi_1' \sim \pi_2'$ if C' be two formulas. Then $(\!|f \wedge g|\!) = (\!|f|\!) \cap (\!|g|\!)$.*

Proof: ($\subseteq$): Let $h \triangleq \pi_1\sigma_1 \sim \pi_2\sigma_2$ *if* $(C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge \sigma_1 \wedge \bigwedge \sigma_2$ be in $f \wedge g$ and let $(\gamma_1, \gamma_2) \in (\!|h|\!)$. There exists a valuation $\rho : \mathit{Var} \to \mathcal{T}$ such that $(\gamma_i, \rho) \models \pi_i\sigma_i \wedge (C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge \sigma_1 \wedge \bigwedge \sigma_2$ for $i = 1, 2$. Since $\pi_i(\sigma_1 \cup \sigma_2) = \pi_i\sigma_i$ for $i = 1, 2$, we also have $(\gamma_i, (\sigma_1 \cup \sigma_2)\rho) \models \pi_i \wedge C$ for $i = 1, 2$. This implies $(\gamma_1, \gamma_2) \in (\!|f|\!)$. Symmetrically, $(\gamma_1, \gamma_2) \in (\!|g|\!)$. This ends the proof of the ($\subseteq$) inclusion.
($\supseteq$): Let $(\gamma_1, \gamma_2) \in (\!|f|\!) \cap (\!|g|\!)$. We show that there is h in $f \wedge g$ s.t. $(\gamma_1, \gamma_2) \in (\!|h|\!)$. From $(\gamma_1, \gamma_2) \in (\!|f|\!)$ we obtain that there is $\eta : \mathit{Var} \to \mathcal{T}$ such that $(\gamma_i, \eta) \models \pi_i \wedge C$ for $i = 1, 2$, and from $(\gamma_1, \gamma_2) \in (\!|g|\!)$ we obtain that there is $\eta' : \mathit{Var} \to \mathcal{T}$ such that $(\gamma_i, \eta') \models \pi_i' \wedge C'$ for $i = 1, 2$. Thus, $\pi_i\eta = \pi_i'\eta' = \pi_i(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}) = \pi_i'(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}})$, and then there exists $\sigma_i \in U(\pi_i, \pi_i')$ and $\rho_i : \mathit{Var} \to \mathcal{T}$ such that $\sigma_i\rho_i = \eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}$ for $i = 1, 2$. Let $\rho \triangleq \rho_1|_{ran(\sigma_1)} \cup \rho_2|_{ran(\sigma_2)} \cup \eta|_{var(f)} \cup \eta'|_{var(g)} \cup \eta'|_R$, where R is the set $\overline{ran(\sigma_1) \cup ran(\sigma_2) \cup var(f) \cup var(g)}$.

Let $h \triangleq \pi_1\sigma_1 \sim \pi_2\sigma_2$ *if* $(C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge \sigma_1 \wedge \bigwedge \sigma_2$. We have

$$\pi_i\sigma_i\rho = \pi_i\sigma_i\rho_i|_{ran(\sigma_i)} = \pi_i\sigma_i\rho_i = \pi_i(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}) = \gamma_i. \tag{9}$$

In order to prove $(\gamma_1, \gamma_2) \in (\!|h|\!)$ we prove $((C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge \sigma_1 \wedge \bigwedge \sigma_2)\rho = \mathit{true}$.

Consider first the term $C(\sigma_1 \cup \sigma_2)$. The variables in it are of one of three forms:

- $x\sigma_1$ for some $x \in var(\pi_1)$. Then, $x(\sigma_1 \cup \sigma_2)\rho = x\sigma_1\rho = x\sigma_1\rho_1 = x(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}) = x\eta|_{var(f)}$;
- $y\sigma_2$ for some $y \in var(\pi_2)$. Then, $y(\sigma_1 \cup \sigma_2)\rho = y\sigma_2\rho = y\sigma_2\rho_2 = y(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}) = y\eta|_{var(f)}$;
- z, for some $z \in var(f) \setminus (var(\pi_1) \cup var(\pi_2))$. Then, $z(\sigma_1 \cup \sigma_2)\rho = z\rho = z\eta|_{var(f)}$.

Thus,

$$C(\sigma_1 \cup \sigma_2)\rho = C\eta|_{var(f)} = \mathit{true} \tag{10}$$

In a similar way we get

$$C'(\sigma_1 \cup \sigma_2)\rho = C'\eta|_{var(g)} = \mathit{true} \tag{11}$$

Note that ρ is not completely symmetrical as η' was chosen to valuate the variables in R, but those variables do not matter anyway. Finally, for $i = 1, 2$:

$$\begin{aligned}
(\bigwedge \sigma_i)\rho &= (\bigwedge_{x \in dom(\sigma_i)} x = x\sigma_i)\rho \\
&= \bigwedge_{x \in dom(\sigma_i)} x\rho = x\sigma_i\rho \\
&= \left(\bigwedge_{x \in var(\pi_i)} x\rho = x\sigma_i\rho\right) \wedge \left(\bigwedge_{x \in var(\pi_i')} x\rho = x\sigma_i\rho\right) \\
&= \left(\bigwedge_{x \in var(\pi_i)} x\eta = x\sigma_i\rho_i\right) \wedge \left(\bigwedge_{x \in var(\pi_i')} x\eta' = x\sigma_i\rho_i\right) \\
&= \mathit{true}
\end{aligned} \tag{12}$$

From (9), (10), (11), and (12) we obtain $(\gamma_1, \gamma_2) \in (\!|h|\!) \subseteq (\!|f \wedge g|\!)$. $\square$

Corollary 2 $[\![f \wedge g]\!] = [\![f]\!] \cap [\![g]\!]$.

Proof: $[\![f \wedge g]\!]$ and $[\![f]\!] \cap [\![g]\!]$ are the subsets of $(\!|f \wedge g|\!)$ and of $(\!|f|\!) \cap (\!|g|\!)$, respectively, which consist of pairs (γ_1, γ_2) such that neither γ_1, γ_2 may lead to deadlocks.□

Theorem 1 (soundness of $\vdash^{\circlearrowleft}$) *Let Γ be a set of formulas such that $(\!|\Gamma|\!) \subseteq (\!|\Omega|\!)$ and for all $g \in \Gamma$, $[\![g]\!] \neq \emptyset$. If $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ then $\mathcal{S} \models \Gamma$.*

Proof: If $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ then there exists a finite proof in $\vdash^{\circlearrowleft}$, of the form $(F_0, G_0) \Longrightarrow \cdots \Longrightarrow (F_n, G_n)$, with $F_0 = \emptyset$, $G_0 = \Gamma$, and $G_n = \emptyset$. We assume that n is the smallest natural number such that $G_n = \emptyset$. The proof amounts to *eliminating* all the goals in Γ, possibly by replacing them by new goals and by adding hypotheses along the way.

Let $\mathcal{F} = \bigcup_{i \leq n} F_i (= F_n)$ and $\mathcal{G} = \bigcup_{i \leq n} G_i$. We now define the relation $\succ$ over $\mathcal{G}$ by $g \succ g'$ iff $\$g < \$g'$, where $\$g$ is the step in which g *is eliminated for the last time* from $\mathcal{G}$; note that goals may re-appear by derivation, and may be re-eliminated again.
We start by noting that for all $g \in \mathcal{G}$

($\clubsuit$) if g is last eliminated by [Derive] then for all $g' \in \Delta(g)$, $g \succ g'$.

Indeed, in the last elimination of g by [Derive], the derivatives of g are added to $\mathcal{G}$, and their last elimination strictly succeeds that of g, which proves ($\clubsuit$).

Since $\$g$ is a natural number less than or equal to n, it follows that $\succ$ is Noetherian. If g is a formula with $\mathcal{S} \vdash g$, then there is a weak bisimulation $R(g)$ that is a witness for $\mathcal{S} \models g$ (since $\vdash$ is sound for $\models$). Let R denote the relation

$$R = \bigcup_{f \in \mathcal{F}, \mathcal{S} \not\vdash f} [\![f]\!] \cup \bigcup_{\mathcal{S} \vdash g} R(g)$$

We now prove that for all $g \in \mathcal{G}$,

($\spadesuit$) if $(\gamma_1, \gamma_2) \in [\![g]\!]$, then there is $(\gamma_1'', \gamma_2'') \in R$ such that $\gamma_1 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_1''$ and $\gamma_2 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_2''$.

We proceed by Noetherian induction on the relation $\succ$. For this, we consider *the last time g was eliminated from $\mathcal{G}$*. There are several cases:

- g is last eliminated by [Reduce]: we have the following two sub-cases (recall that $\vdash$ is sound for $\models$):
  - $\mathcal{S} \models g$, which implies $[\![g]\!] \subseteq R(g) \subseteq R$, and the property ($\spadesuit$) is obtained by taking $\gamma_1'' = \gamma_1$, $\gamma_2'' = \gamma_2$.
  - $(\!|g|\!) \subseteq (\!|\mathcal{F}|\!)$: Then there is $i \leq n$ such that $(\!|g|\!) \subseteq (\!|F_i|\!)$. Using Lemma 1 we get $[\![g]\!] \subseteq [\![F_i]\!] \subseteq [\![\mathcal{F}]\!] \subseteq R$, and the property ($\spadesuit$) is obtained by by taking $\gamma_1'' = \gamma_1$, $\gamma_2'' = \gamma_2$.
- g is last eliminated by [Derive]: then, $\Delta(g) \neq \emptyset$. Assume $\Delta^l(g) \neq \emptyset$ (the case $\Delta^r(g) \neq \emptyset$ is similar). If $(\gamma_1, \gamma_2) \in [\![g]\!]$ then there is $g' \in \Delta^l(g)$ and $(\gamma_1', \gamma_2) \in [\![g']\!]$ s.t. $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1'$ by Lemma 3. Using ($\clubsuit$) we obtain $g \succ g'$, thus, ($\spadesuit$) holds for g' by the induction hypothesis: there is $(\gamma_1'', \gamma_2'') \in R$ such that $\gamma_1' \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_1''$ and $\gamma_2 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_2''$. Hence $\gamma_1 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_1''$ and $\gamma_2 \Rightarrow^{*\mathcal{T}}_{\mathcal{S}} \gamma_2''$, which proves ($\spadesuit$) for g.

The proof of ($\spadesuit$) is now complete. Next, we prove

($\diamondsuit$) for each $g \in \Gamma$, $[\![g]\!] \subseteq R$.

For this, we consider two cases, according to how g was first eliminated.

- If g was first eliminated by [Reduce], then, again, we have two sub-cases:
  - $\mathcal{S} \models g$ and thus $[\![g]\!] \subseteq R(g) \subseteq R$ by the definition of R, which proves $(\diamondsuit)$ in this case;
  - there is $i \leq n$ such that $(\![g]\!) \subseteq (\![F_i]\!)$. Using Lemma 1, $[\![g]\!] \subseteq [\![F_i]\!] \subseteq [\![\mathcal{F}]\!] \subseteq R$, which proves $(\diamondsuit)$ in this case;
- If g was first eliminated by [Derive]: since $(\![g]\!) \subseteq (\![\Omega]\!)$ (recall that $(\![\Gamma]\!) \subseteq (\![\Omega]\!)$), it follows that for each $(\gamma_1, \gamma_2) \in (\![g]\!)$ there is $h \in \Omega$ such that $(\gamma_1, \gamma_2) \in (\![g]\!) \cap (\![h]\!) = (\![g \wedge h]\!)$. Hence, $(\![g]\!) \subseteq (\![g \wedge \Omega]\!)$ and using Lemma 1, $[\![g]\!] \subseteq [\![g \wedge \Omega]\!] \subseteq [\![F']\!] \subseteq [\![\mathcal{F}]\!] \subseteq R$. The proof of $(\diamondsuit)$ is done.

By hypothesis of our theorem, all goals $g \in \Gamma$ satisfy $[\![g]\!] \neq \emptyset$. To reach our conclusion we only need to prove that R is a weak bisimulation.

For this, we first note that $R \subseteq (\![\Omega]\!)$ as all the goals g such that $\mathcal{S} \vdash g$ contribute to R with a weak bisimulation relation $R(g)$, which by definition of weak bisimulation satisfies $R(g) \subseteq (\![\Omega]\!)$; and that for all $f \in \mathcal{F}$ such that $\mathcal{S} \not\vdash f$, $[\![f]\!] \subseteq]\!]\Omega]\!]$ because we have $g \wedge \Omega \vdash f$ for certain g and hence $[\![f]\!] \subseteq]\!]g \wedge \Omega]\!] = [\![g]\!] \cap [\![\Omega]\!] \subseteq [\![\Omega]\!]$. Thus, $R \subseteq (\![\Omega]\!)$ holds.

The remaining conditions for R being a weak bisimulation are established as follows. Let $(\gamma_1, \gamma_2) \in R$. If $(\gamma_1, \gamma_2) \in R(g)$ for some $g \in \Gamma$ with $\mathcal{S} \vdash g$ then the weak bisimulation conditions for (γ_1, γ_2) hold as $R(g)$ is a weak bisimulation. Hence, we assume $(\gamma_1, \gamma_2) \in \bigcup_{f \in \mathcal{F}, \mathcal{S} \not\vdash} [\![f]\!]$. Let γ_1' such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$. If $\gamma_1 = \gamma_1'$ then all the successors of γ_1 are equal to γ_1; using $(\spadesuit)$ we obtain that there is γ_2'' such that $(\gamma_1', \gamma_2'') \in R$, thus, R is a weak bisimulation. Hence, we assume $\gamma_1 \neq \gamma_1'$. Since $(\gamma_1, \gamma_2) \in \bigcup_{f \in \mathcal{F}, \mathcal{S} \not\vdash} [\![f]\!]$, there is $f \in \mathcal{F}$ with $\mathcal{S} \not\vdash f$ such that $(\gamma_1, \gamma_2) \in [\![f]\!]$. Now, for every such f, there is a formula g with $\emptyset \neq \Delta(g) \in \mathcal{G}$ such that $g \wedge \Omega \vdash f$ by the [Derive] rule. It follows that $(\gamma_1, \gamma_2) \in [\![g]\!]$ (here we use the properties of the $\wedge$ operation: by Corollary 2, $[\![g \wedge \Omega]\!] \subseteq [\![g]\!]$, thus, if $(\gamma_1, \gamma_2) \in [\![f]\!]$ with $(\![f]\!) \subseteq (\![g \wedge \Omega]\!)$ then $(\gamma_1, \gamma_2) \in [\![g]\!]$).

By Corrolary 1, there is $g' \in \Delta(g)$ such that $(\gamma_1', \gamma_2) \in [\![g']\!]$. Using $(\spadesuit)$ for g' and (γ_1', γ_2) we obtain $(\gamma_1'', \gamma_2'') \in R$ such that $\gamma_1' {\Rightarrow^*}_{\mathcal{S}}^{\mathcal{T}} \gamma_1''$ and $\gamma_2 {\Rightarrow^*}_{\mathcal{S}}^{\mathcal{T}} \gamma_2''$. This proves the first condition for R to be a weak bisimulation.

The second condition is proved similarly. This concludes the proof of our theorem. □

Regarding weak completeness We now prove the weak completeness theorem. First, we note that with the updated notion of derivatives in which formulas with unsatisfiable are removed the results proved so far on derivatives still hold:

1. Lemma 2 holds because whenever there exists $g' \in \Delta^l(g)$ such that $(\gamma_1', \gamma_2) \in (\![g']\!)$, $(\![g']\!) \neq \emptyset$, hence, it is enough to consider derivatives with satisfiable conditions;
2. Corollary 1 holds because it only uses the properties of derivatives stated in Lemma 2;
3. Lemma 3 holds because it only uses the properties of derivatives in Corollary 1.

Thus, soundness (Theorem 1) still holds with the new definition of derivatives.

Lemma 5 *If $(\![f]\!) \neq \emptyset$ then for all $f' \in \Delta(f)(= \Delta^l(f) \cup \Delta^r(f))$, $(\![f']\!) \neq \emptyset$*

Proof: Directly from the new definition of derivatives (without unsatisfiable formulas).

Lemma 6 *If $(\gamma_1', \gamma_2') \in (\![\Delta^l(g)]\!)$ then there is $(\gamma_1, \gamma_2') \in (\![g]\!)$ such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$.*

Proof: Let $g : \pi_1 \sim \pi_2$ *if* C. Now, $(\gamma_1', \gamma_2') \in (\![\Delta^l(g)]\!)$ implies $(\gamma_1', \gamma_2') \in (\![g']\!)$ for some $g' \in (\![\Delta^l(g)]\!)$, and g' is of the form $g' : r\sigma \sim \pi_2$ *if* $(C \wedge C')\sigma \wedge \bigwedge\sigma$ for some rewrite rule $l \wedge C' \Rightarrow r \in \mathcal{S}$ and unifier $\sigma \in U(\pi_1, l)$. Hence, there exists $\rho'' : \mathit{Var} \to \mathcal{T}$ such that:

- $\gamma'_1 = r\sigma\rho''$
- $\gamma'_2 = \pi_2\rho''$
- $(C \wedge C')\sigma\rho'' = \mathit{true}$.

Let $\gamma_1 = l\sigma\rho''$. Then, $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'_1 = r\sigma\rho''$ by applying the rule $l \wedge C' \Rightarrow r$ with valuation $\sigma\rho''$ on γ_1 (note that $C'\sigma\rho'' = \mathit{true}$). Moreover, $\sigma \in U(\pi_1, l)$, which means $l\sigma = \pi_1\sigma$ and implies $l\sigma\rho'' = \pi_1\sigma\rho''$. Since $C\sigma\rho'' = \mathit{true}$, and $\gamma'_2 = \pi_2\rho'' = \pi_2\sigma\rho''$ (the last equality because $\mathit{dom}(\sigma) = \mathit{var}(\pi_1) \cup \mathit{var}(l)$, whose intersection with $\mathit{var}(\pi_2)$ is empty, cf. Remarks 1 and 2), $(\gamma_1, \gamma'_2) \in (\!|g|\!)$, and the proof is done.

Theorem 2 (weak completeness of $\vdash^{\circlearrowleft}$) *Assume $\mathcal{S} \models \Gamma$ and the proof system $\vdash^{\circlearrowleft}$ terminates on Γ. Then, $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$.*

Proof: By contradiction: assume $\vdash^{\circlearrowleft}$ terminates but $\mathcal{S} \not\vdash^{\circlearrowleft} \Gamma$. This may only happen when $\vdash^{\circlearrowleft}$ encounters a formula g it cannot eliminate. In particular, this means $\Delta(g) = \emptyset$.

1. if $(\!|g|\!) = \emptyset$ then using Lemma 5 we obtain an initial goal g_0 with $(\!|g_0|\!) = \emptyset$. But $\mathcal{S} \models \Gamma$ implies for all $g_0 \in \Gamma$, $[\![g_0]\!] \neq \emptyset$ and thus $(\!|g_0|\!) \neq \emptyset$: a contradiction.

2. thus, $(\!|g|\!) \neq \emptyset$. Let $g : \pi_1 \sim \pi_2 \ \mathit{if}\ C$. Assume first that both programs in π_1 and π_2 are empty; then, for all $(\gamma_1, \gamma_2) \in (\!|g|\!)$, γ_1, γ_2 are final configurations, and $(\!|g|\!) = [\![g]\!] \neq \emptyset$.

   (a) if $(\!|g|\!) \subseteq (\!|\Omega|\!)$, then using **Assumption 4** we obtain $\mathcal{S} \vdash g$, hence, g can be eliminated by [Reduce], in contradiction with the hypothesis that g cannot be eliminated.

   (b) thus, $(\!|g|\!) \not\subseteq (\!|\Omega|\!)$: then, there exists $(\gamma_1, \gamma_2) \in (\!|g|\!) \setminus (\!|\Omega|\!)$. Using Lemma 6 we obtain there exists an initial goal $g_0 \in \Gamma$ and instances $(\gamma'_1, \gamma'_2) \in (\!|g_0|\!)$ such that $\gamma'_1 {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_1$ and $\gamma'_2 {\Rightarrow^*}^{\mathcal{T}}_{\mathcal{S}} \gamma_2$. Moreover, $(\gamma'_1, \gamma'_2) \in [\![g_0]\!]$ because infinite computations start in γ'_1, γ'_2 (which self-loop in γ_1, γ_2 respectively). We claim $\mathcal{S} \not\models g_0$. Indeed, if $\mathcal{S} \models g_0$ then there is a weak bisimulation $R \supseteq [\![g_0]\!]$. Using the properties of weak bisimulation we obtain that $(\gamma_1, \gamma_2) \in R$. But this is impossible since $(\gamma_1, \gamma_2) \notin (\!|\Omega|\!)$. Thus, we have obtained a contradiction with the hypothesis $\mathcal{S} \models \Gamma$.

3. the assumption that that both programs in π_1 and π_2 are empty lead to contradiction, thus, at least one of the programs in π_1 and π_2 is nonempty. Then for all $(\gamma_1, \gamma_2) \in (\!|g|\!)$, at least one of γ_1, γ_2 is a deadlock. Using **Assumption 4** we obtain $(\gamma_1, \gamma_2) \notin (\!|\Omega|\!)$, and we obtain a contradiction with the hypothesis $\mathcal{S} \models \Gamma$ just like in case 2(b) above.

The assumption $\mathcal{S} \not\vdash^{\circlearrowleft} \Gamma$ leads to contradictions, which means $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$. □

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399