Program Equivalence by Circular Reasoning

Dorel Lucanu[1] and Vlad Rusu[2]

[1] Al. I. Cuza University of Iaşi, Romania dlucanu@info.uaic.ro
[2] Inria Lille Nord-Europe, France Vlad.Rusu@inria.fr

Abstract. We propose a deductive system for automatically proving the equivalence of programs written in deterministic languages that have a formal, term-rewriting based operational semantics. Symbolic programs (i.e., programs in which some statements or expressions are symbolic variables) can also be proved equivalent using the proposed approach. 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 the deductive system is suitable for proving equivalence both for terminating and non-terminating programs. We also report on a prototype implementation of the proposed system in the $\mathbb{K}$ Framework.

1 Introduction

In this paper we propose a formal notion of program equivalence together with a deductive system for proving it. Programs can belong to any deterministic language whose semantics is specified by a set of rewrite rules. The equivalence is weak bisimulation, allowing several instructions of one program to be matched by several instructions of the other one. The deductive 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, it correctly solves the program-equivalence problem thanks to its soundness and weak completeness properties. These properties are informally presented below and are formalised 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.

Example 1. The following programs (referred to by *for* and *while*) are symbolic:

```
for I from A to B do
  {
    S
  }
```

```
I = A ;
while I <= B do {
    S ;
    I = I + 1
  }
```

Their symbolic variables I, A, B, S can be matched by, respectively, any identifier (I), arithmetical expression (A, B), and program statement (S). We shall use this

example in the paper and assume that the statement S is terminating. A concrete instance of the *for* symbolic program is e.g., `for k from 1 to i+3 do {j=j+1}`. 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, $\mathcal{L}$. We note that currently, our approach works only for deterministic languages;
2. extending the semantics of $\mathcal{L}$ to symbolic statements and expressions;
3. running the deductive system over the extended semantics to check the equivalence of programs in $\mathcal{L}$.

Running the deductive system amounts essentially to symbolically executing the language's semantics. This leads to one of the following three possible 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.

Example 2. The operational semantics of an imperative language including *for* and *while* statements such as those in Example 1, is given in Section 2. Running the semantics consists in transforming *configurations*, which are pairs built from:

- the programs that remains to be executed, and
- a mapping of variables to values denoting the current state.

For example, starting from a configuration $\langle$`for` I `from` A `to` B `do { }` S `}`$, state\rangle$ after a few semantical steps one reaches a configuration $\langle P_1, state_1\rangle$, where either

- P_1 is the empty program, and
- $state_1$ is the effect of I `=` A on $state$, and $state_1(I) > state_1(B)$

or

- P_1 is `for` I `from` A `+ 1 to` B `do { }` S `}`, and
- $state_1$ is the effect of the sequence I`=`A`;` S on $state$, and $state_1(I) \leq state_1(B)$.

Similarly, starting from $\langle I$ `=` A `; while` I `<=` B `do { }` S `;` I `=` I `+ 1 }`$, state\rangle$, after a few semantical steps, one reaches a configuration $\langle P_2, state_2\rangle$, where either

- P_2 is the empty program, and
- $state_2$ is the effect of I `=` A on $state$, and $state_2(I) > state_2(B)$

or

- P_2 is I `=` A `+ 1 ; while` I `<=` B `do { }` S `;` I `=` I `+ 1 }`
- $state_2$ is the effect of the sequence I`=`A`;` S on $state$, and $state_2(I) \leq state_2(B)$.

That is, after some transformations of the original configurations, one has either:

- equal configurations, with empty programs (those in the left-hand side), or
- configurations with nonempty programs (in the right-hand side) which are, in some sense defined in the paper, *instances* of the original configurations.

At this point, our deductive system terminates and reports that the two initial configurations are equivalent. Intuitively, this is because it "realises" the programs can only behave in similar ways from there on in the future, forever reproducing the behaviours exhibited after the first few semantic steps. Thus, even though the symbolic executions do not terminate - since the *for* and *while* loops do not have finite bounds - our proof system terminates, here, with success.

A straightforward extension of our program-equivalence approach is proving the equivalence between programs of *two* different languages, say, P_1 of language $\mathcal{L}_1$ and P_2 of language $\mathcal{L}_2$. This problem can be reduced to proving the equivalence of P_1, P_2, both seen as programs of a disjoint language-union $\mathcal{L}_1 \uplus \mathcal{L}_2$ (i.e. some renaming of configuration syntax and semantical rules may be necessary in order to avoid confusion between the definitions of the two languages). Moreover, the semantical correctness of syntax-directed translations between languages can be cast as a (symbolic) program-equivalence problem: if each each basic instruction pattern of $\mathcal{L}_1$ is proved equivalent to its translation in $\mathcal{L}_2$ then the whole translation of $\mathcal{L}_1$ to $\mathcal{L}_2$ is semantically correct. For instance, our for-to-while-loops example can be seen as part of a translation from a language that has for-loops to a language that has while-loops. We leave such applications for future work.

Our main contribution is thus a proof system for program equivalence that is suitabe both for concrete and symbolic programs and for terminating and non-terminating ones, together with its soundness and weak completeness results.

The rest of the paper is organised as follows. Section 2 presents our running example: IMP, a simple imperative language and its semantics in $\mathbb{K}$ [1]. $\mathbb{K}$ is a formal framework for defining operational semantics of programming languages.

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

Section 4 contains our proposed definition for program equivalence, together with the rationale for choosing this definition among several possible ones.

Section 5 gives the syntax and semantics of a logic for program 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 $\mathbb{K}$.

Int ::= domain of integer numbers (including operations)
$Bool$ ::= domain of boolean constants (including operations)
Id ::= domain of identifiers

$AExp ::= Int \mid Id$
$\quad \mid AExp$ `/` $AExp$ [strict]
$\quad \mid AExp$ `*` $AExp$ [strict]
$\quad \mid AExp$ `+` $AExp$ [strict]
$\quad \mid$ `(` $AExp$ `)`

$BExp ::= Bool$
$\quad \mid AExp$ `<=` $AExp$ [strict]
$\quad \mid$ `not` $BExp$ [strict]
$\quad \mid BExp$ `and` $BExp$ [strict(1)]
$\quad \mid$ `(` $BExp$ `)`

$Stmt ::=$ `skip` $\mid Stmt$ `;` $Stmt$
$\quad \mid Id$ `=` $AExp$
$\quad \mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$ [strict(1)]
$\quad \mid$ `{` $Stmt$ `}`
$\quad \mid$ `while` $BExp$ `do` $Stmt$
$\quad \mid$ `for` Id `from` $AExp$ `to` $AExp$ `do` $Stmt$ [strict(2,3)]

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

Fig. 1. $\mathbb{K}$ Syntax of IMP

The conclusion, related work, and future work are presented in Section 9. Formal proofs for all the results in the paper are given in the extended version, currently available at `http://fmse.info.uaic.ro/publications/155/`.

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 [1]. 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 subcell `k` and a subcellcell `env` (see Figure 2). For other languages the configuration structure may be different.

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

Fig. 2. $\mathbb{K}$ Configuration of IMP

$\langle\langle I_1$ `+` I_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I_1$ `*` I_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{Int} I_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I_1$ `/` I_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I_1$ `<=` I_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{Int} I_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle true$ `and` B $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle false$ `and` B $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle false$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`not` $true$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle false$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`not` $false$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle true$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`skip` $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle S_1; S_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`{` S `}` $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`if` $true$ `then` S_1 `else` S_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`if` $false$ `then` S_1 `else` S_2 $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2$ $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`while` B `do` S $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow$

$\langle\langle$`if` B `then{` S `;while` B `do` S `}else skip` $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle$`for` X `from` I_1 `to` I_2 `do` S $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow$

$\langle\langle X$ `=` I_1 `;if` X `<=` I_2 `then{` S `;for` X `from` I_1 `+` 1 `to` I_2 `do` S `}else skip` $\cdots\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle X$ $\cdots\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle$`lookup` (Env, I) $\cdots\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}$ $\cdots\rangle_{\mathsf{cfg}}$

$\langle\langle X$ `=` I $\cdots\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}$ $\cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle$ $\cdots\rangle_{\mathsf{k}}\langle$`update` $(Env, X, I)\rangle_{\mathsf{env}}$ $\cdots\rangle_{\mathsf{cfg}}$

Fig. 3. $\mathbb{K}$ Semantics of IMP

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 exactly one rule.

In Figure 3, the operations `lookup`: $Map \times Id \to Int$ and `update`: $Map \times Id \times Int \to Map$ 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 $BExp \setminus \{false, true\}$, B ranges over $\{false, true\}$, 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 basic concepts of algebraic specification and term rewriting. We consider fixed a language $\mathcal{L}$ defined by the following:

1. A many-sorted algebraic signature Σ that includes a distinguised sort *Cfg* for configurations. The signature Σ also includes a sort *Bool* with the usual constants and operations as well as other basic sorts (integers, maps, ...). The syntax of the language $\mathcal{L}$ is also defined in Σ. Let T_Σ denote the initial Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s.
2. A sort-wise infinite set of variables *Var*. Let $T_\Sigma(Var)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(Var)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.
3. A Σ-algebra $\mathcal{T}$. Let $\mathcal{T}_s$ denotes the elements of $\mathcal{T}$ of sort s; if $s = Cfg$ then these elements are called *configurations*. Any *valuation* $\rho : Var \to \mathcal{T}$ is extended to a Σ-algebra morphism $\rho : T_\Sigma(Var) \to \mathcal{T}$. If $b \in T_{\Sigma,Bool}(Var)$ then we write $\rho \models b$ iff $b\rho = true$ (the equality is that of $\mathcal{T}$). Equality of terms $t_1, t_2 \in T_\Sigma(Var)$ is defined by $t_1 = t_2$ iff $t_1\rho = t_2\rho$ for all valuations ρ.

We explain these concepts on running example. Each nonterminal from the syntax (*Int*, *Bool*, *AExp*, ...) is a sort in Σ. Each production from the syntax defines an operation in Σ; for instance, the production $AExp ::= AExp\ \texttt{+}\ AExp$ defines the operation $\_\texttt{+}\_ : AExp \times AExp \to AExp$. For the configuration sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map \to Cfg$. The expression $\langle\langle X \texttt{=} I \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto 0\ Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ 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* (the rest of the environment).

The algebra $\mathcal{T}$ interprets *Int* as the set of integers, the operations like $+_{Int}$ (cf. Figure 3) as the corresponding usual operation on integers (the $_{Int}$ underscore is there to distinguish it from the + operation on expressions); *Bool* is interpreted as the set of Boolean values $\{false, true\}$, operations like $\wedge$ as the usual Boolean operations, the sort *Map* as the set of the multisets of bindings of the form $I \mapsto V$, where I ranges over identifiers and V over the integers.

The pieces of code as interpreted by $\mathcal{T}$ as ground terms. For instance, each IMP program is both a term of sort *Stmt* and an element in $\mathcal{T}_{Stmt}$.

Definition 1 (pattern). *A* pattern *is an expression of the form $\pi \wedge b$, where $\pi \in T_{\Sigma,Cfg}(Var)$ and $b \in T_{\Sigma,Bool}(Var)$ and $var(b) \subseteq var(t)$. If $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$, then $(\gamma, \rho) \models \pi \wedge b$ if $\gamma = \pi\rho$ and $\rho \models b$.*

We identify terms $\pi \in T_{\Sigma,Cfg}(Var)$ with paterns $\pi \wedge true$. Sample patterns are

$\langle\langle I_1 \ / \ I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0$ and

$\langle\langle \texttt{for}\ I\ \texttt{from}\ A + 1\ \texttt{to}\ B\ \texttt{do}\ \{\ S\ \} \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$.

A *final configuration* is a configuration where the program component is empty (i.e., there is nothing left to execute). For example, $\langle\langle\cdot\rangle_{\mathsf{k}}\langle A \mapsto 3\rangle_{\mathsf{Env}}\rangle_{\mathsf{cfg}}$ is final.

Definition 2 (semantical rule and transition relation). *A* rule *is a pair of patterns of the form $l \wedge b \Rightarrow r$ (note that r is in fact the pattern $r \wedge true$). Let $\mathcal{S}$ be a set of rules meant to define the operational semantics of the language $\mathcal{L}$. The semantics $\mathcal{S}$ defines a transition relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ given by $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ iff, either*

- *there are $l \wedge b \Rightarrow r$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$;*
- *or γ is a final configuration, and $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma$.*

The introduction of self-loops on final configurations is a technical point, only required for proving the soundness of our program-equivalence deductive system. A configuration γ is a *deadlock* if there is no configuration γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$. The relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ is *deterministic* if it is the graph of a partial function from $\mathcal{T}_{Cfg}$ to itself.

Assumption 1 *In this paper we assume that the relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ is deterministic.*

We shall be using unification in our program-equivalence deductive system. A *unifier* of two terms t_1, t_2 is a substitution σ such that $t_1\sigma = t_2\sigma$.

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* most general unifiers *π and l, which satisfy the property that for all unifiers ρ of π and l, there exist substitutions $\sigma \in U(\pi, l)$ and η such that $\sigma\eta = \rho$.*

We noticed that in practice the most general unifiers can often be found by matching with a slightly different set of rules $\mathcal{S}'$ such that $\Rightarrow_{\mathcal{S}'}^{\mathcal{T}} \ = \ \Rightarrow_{\mathcal{S}}^{\mathcal{T}}$. (The general transformation from $\mathcal{S}$ to $\mathcal{S}'$ will be presented in a different paper.) This is important for implementation purposes, since we implement program equivalence in $\mathbb{K}$, which has matching but no unification. We illustrate below how this can be done via an example; other examples follow later in the paper.

Example 3. 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 the rule $\langle\langle(\texttt{if}\ true\ \texttt{then}\ S_1'\ \texttt{else}\ S_2') \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1' \curvearrowright S\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. 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 $\sigma = (B \mapsto true, S_1' \mapsto S_1, S_2' \mapsto S_2, M' \mapsto M)$. 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

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

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 modified rule's condition $B' = true$, amount to the above most general unifier σ.

4 Defining Program Equivalence

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

A natural approach would consist in using *strong bisimulation*: a relation $R \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is a strong bisimulation if for all $(\gamma_1, \gamma_2) \in R$, whenever $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$, there is a transition $\gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2'$ such that $(\gamma_1', \gamma_2') \in R$, and, symmetrically, whenever $\gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2'$, there is a transition $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ 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 weaken strong bisimulation. A relation $R \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is a *weak bisimulation* if for all $(\gamma_1, \gamma_2) \in R$, whenever $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$, there is a (possibly empty) transition sequence $\gamma_2 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2'$ such that $(\gamma_1', \gamma_2') \in R$, and, symmetrically, whenever $\gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2'$, there is a (possibly empty) transition sequence $\gamma_1 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1'$ such that $(\gamma_1', \gamma_2') \in R$. That is, we have generalised the one step $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ to any number of steps $\Rightarrow_{\mathcal{S}}^{*\mathcal{T}}$ (zero, one, or several). But this does not work either, since the universal relation $\mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is a weak bisimulation in the above sense.

Thus, we need in general upper bounds on weak bisimulations. We fix a relation $O \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$, which we call the *observation relation*, and define O-weak bisimulation to be a weak bisimulation R such that $R \subseteq O$.

Example 4. Consider the $\mathbb{K}$ definition of IMP. Natural candidates for the relation O are obtained by letting $(\langle\langle P_1 \rangle_{\mathsf{k}} \langle Env_1 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}, \langle\langle P_2 \rangle_{\mathsf{k}} \langle Env_2 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}) \in O$ whenever a chosen subset of the variables occurring in P_1, P_2 are mapped to the same values in Env_1, Env_2 respectively. We call the chosen set of variables *observers*.

Actually, it can be proved that the above notion of O-weak bisimulation is equivalent, under our determinism assumption, to the following one:

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* $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ *then* $\gamma_1 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1''$ *and* $\gamma_2 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2''$, *for some* $(\gamma_1'', \gamma_2'') \in R$
- *if* $\gamma_2 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_2'$ *then* $\gamma_1 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1''$ *and* $\gamma_2' \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2''$ *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 5. The following 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 observer `x`. 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, implemented using observers or otherwise, gives us quite a lot of expressiveness for capturing various kinds of program equivalences. For example, a usual form of equivalence is: two programs are equivalent if, whenever presented with the same input, if they terminate they produce the same output (see, e.g., [2,3]). 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.

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

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

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 $[\![f]\!]$, is the subset of $(\!|f|\!)$ containing pairs configurations that are origins of computations that are either finite and ending in a final configuration, or infinite. Since the transition relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ has self-loops on final configurations, this amounts to the following definition:

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*
$[\![f]\!] \triangleq \{(\gamma_1, \gamma_2) \in (\!|f|\!) \mid \forall j \in \mathbb{N}. \forall i \in \{1, 2\}. \exists \gamma_i^j \in \mathcal{T}_{Cfg}. \gamma_i = \gamma_i^0 \wedge \gamma_i^j \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_i^{j+1}\}$.

Example 7. For the formula from Example 7, the $(\!|\cdot|\!)$ semantics contains all pairs of concrete configurations (γ_1, γ_2) obtained by replacing the variables A, B, I, S by ground terms of the respective sorts. The $[\![\cdot]\!]$ semantics is the subset of $(\!|\cdot|\!)$ such that, for $i = 1, 2$ the computation starting in γ_i (unique, by the determinism assumption) either reaches a final configuration or does not terminate.

We now define what it means for a formula f to be *valid*. Intuitively, we want to capture the notion 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 only interested in equivalences for configurations γ_1, γ_2 that are origins of computations that reach final configurations or do not terminate. That is, by definition, if γ_1 or γ_2 terminate improperly (in a deadlock, obtained, e.g., by attempting to perform a division by zero) then $\gamma_1 \not\sim \gamma_2$.

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

The condition $[\![f]\!] \neq \emptyset$ is added to ensure that a formula is not *vacuously* valid.

In Section 7 we give a proof system for proving the validity of formulas.

6 Auxiliary Operations: Derivatives and Conjunction

6.1 Derivatives

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 $(l \wedge C' \Rightarrow r) \in \mathcal{S}$, $\sigma^l \in U(\pi_1, l)$, $\sigma^r \in U(\pi_2, r)$:*

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

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 8. 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 derivatives:

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

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

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

where the conjuncts $B = \mathit{true}$ and $B = \mathit{false}$ are induced by the most general 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 equalities $S_1' = S_1$, $S_2' = S_2$, $M' = M$ were removed from conditions since the are superfluous, as S_1', S_2', and M' do not occur in the rest of the formula.

Similarly, f has two right-derivatives, i.e., $\Delta^r(f)$ are the formulas in the set

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

Derivatives have the following properties, which we use for proving the soundness of our proof system. Let f be a formula. For all $(\gamma_1, \gamma_2) \in [\![f]\!]$:

- if $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ and $\gamma_1 \neq \gamma_1'$, there exists $f' \in \Delta^l(f)$ such that $(\gamma_1', \gamma_2) \in [\![f']\!]$;
- if $\Delta^l(f) \neq \emptyset$ there is $f' \in \Delta^l(f)$ and $(\gamma_1', \gamma_2) \in [\![f']\!]$ such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$;
- and two properties symmetrical to the above ones regarding right-derivatives.

The properties say that each semantical step in a component of a configuration pair is mirrored by a derivative computation, and, conversely, that each derivative computation corresponds to a semantical step. Thus, the symbolic computations performed by the derivatives of formulas faithfully reproduce the semantical steps over the configuration pairs denoted by those formulas.

6.2 Conjunction

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* $f \wedge g = \{\pi_1\sigma_1 \sim \pi_2\sigma_2 \;\mathit{if}\; (C \wedge C')(\sigma_1 \cup \sigma_2) \wedge \bigwedge\sigma_1 \wedge \bigwedge\sigma_2 \mid \sigma_1 \in U(\pi_1, \pi_1'),\, \sigma_2 \in U(\pi_2, \pi_2')\}$.

The above definition assumes that $U(\pi_1, \pi_1')$ and $U(\pi_2, \pi_2')$ exist.When we use conjunction in the proof system for validity we will make sure this is the case.

Conjunction has the following properties, which are also used for proving the soundness of our proof system: for all formulas f, g: $(\!|f \wedge g|\!) = (\!|f|\!) \cap (\!|g|\!)$ and $[\![f \wedge g]\!] = [\![f]\!] \cap [\![g]\!]$. Thus, conjunction denotes intersection in both semantics.

Example 9. Let f be the formula in Example 8 and let g denote the formula $\langle\langle P_1\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle M''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ $\mathit{if} M' = M''$. 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 $(\!|f \wedge g|\!) = (\!|f|\!) \cap (\!|g|\!) = (\!|f|\!)$.

Example 10. Let $f \triangleq (\langle\langle X = Y\rangle_{\mathsf{k}}\langle \mathtt{update}\,(M,Y,I)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle X = Y\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}})$ and g be as in Example 9. Let again π_1, π_1' and π_2, π_2' denote their left and right-hand sides, respectively. Each of the sets $U(\pi_1, \pi_1')$ and $U(\pi_2, \pi_2')$ consists of a unique substitution: $\sigma_1 = (P_1 \mapsto (X = Y), M' \mapsto \mathtt{update}\ (M,Y,I))$ and respectively $\sigma_2 = (P_2 \mapsto (X = Y), M'' \mapsto M)$. By removing some superfluous conjuncts in the conditions, we get $f \wedge g = (\langle\langle X = Y\rangle_{\mathsf{k}}\langle \mathtt{update}\ (M,Y,I)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle X = Y\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *if* $M' = M \wedge M' = \mathtt{update}\ (M,Y,I)$. That is, $f \wedge g$ is f enforced with the condition $M = \mathtt{update}\ (M,Y,I)$ imposed by g. Again, this is consistent with the fact that conjunction is, semantically speaking, intersection.

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* [4], a coinductive proof technique for infinite data structures and coalgebras of expressions [5].

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 most general 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 $\{g \wedge h \mid h \in \Omega\}$.

We first briefly explain the rules, then state the correctness results, and conclude the section with an example showing how the proof system works on for proving the equivalence of the *for* and *while* programs shown in the introduction.

[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.

That is, any "part" of a goal g that is "included" the observation relation O (remember that $(\!|\Omega|\!) = O$) can be added as hypothesis. In particular, if $\Omega \models g$, meaning that all instances of g satisfy O, then g itself can be added as a hypothesis. The reason why only the "parts" of goals that are included in O can be added as hypotheses is that the set of hypotheses F accumulated during a circular proof denote a certain weak bisimulation that needs to be included in O. This is an essential fact used for proving the soundness of our deductive system.

7.1 Soundness

The following theorem says that $\vdash^{\circlearrowleft}$ is sound for formula validity:

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 $g \in \Gamma$, $[\![g]\!] \neq \emptyset$ is made for the same reason. 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 $\mathcal{S}, \emptyset \vdash^{\circlearrowleft} \Gamma$.

7.2 Weak Completeness

We show in this section 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; indeed, in this case, the proof system terminates because the derivatives (in the [Derive] rule, the only one that can contribute to nontermination) faithfully reproduce the semantical steps and thus also may also only exist in finite number.

Adapting Definitions of Derivatives 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 the following assumptions. The first one says that non-derivable goals g (i.e.,$\Delta(g) = \emptyset$) that denote observationally equivalent configuration pairs ($(\!|g|\!) \subseteq \Omega$) are valid, and are discharged by the basic entailment $\vdash$. The second one says that deadlocks are not observationally equivalent to any other configuration.

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 $(\gamma_1, \gamma_2) \notin (\!|\Omega|\!)$.*

Under these additional assumptions, the derivatives have the following additional properties. If $(\!|f|\!) \neq \emptyset$ then for all $f' \in \Delta(f)(= \Delta^l(f) \cup \Delta^r(f))$, $(\!|f'|\!) \neq \emptyset$, which says that satisfiable formulas have satisfiable derivatives (semantically speaking); and for all $(\gamma_1', \gamma_2') \in (\!|\Delta^l(f)|\!)$ there is $(\gamma_1, \gamma_2') \in (\!|f|\!)$ such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ and $\gamma_1 \neq \gamma_1'$, which says that configurations denoted by a (left)-derivative formula are obtained from configurationss denoted by the derived formula by a semantical step in the left component. A symmetrical result holds for the right component.

These additional properties of derivatives, induced by the above additional assumptions, allow us to prove the weak completeness theorem below.

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

Some explanations: 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 the [Axiom] rule). The proof system may also *terminate unsucessfully* when it generates a finite tree and cannot expand it any more (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 11. 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:

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

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):

$$\langle\langle\rangle_{\mathsf{k}}, \langle \mathtt{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \;\sim\; \langle\langle\rangle_{\mathsf{k}}, \langle \mathtt{update}(M, I, A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \; \mathit{if}\, A >_{Int} B \tag{2}$$

and

$$\begin{array}{l}\langle\langle \mathtt{for}\ I\ \mathtt{from}\ A +_{Int} 1\ \mathtt{to}\ B\ \mathtt{do\{}\ S\ \mathtt{\}}\rangle_{\mathsf{k}}, \langle \mathtt{execute}(S, \mathtt{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \quad \sim \\ \langle\langle I = A +_{Int} 1 \mathtt{;while}\ I <= B\ \mathtt{do\{}\ S \mathtt{;} I = I + 1\ \mathtt{\}}\rangle_{\mathsf{k}}, \langle \mathtt{execute}(S, \mathtt{update}(M, I, A))\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \quad \mathit{if}\ \ \mathtt{not\ occurs(}\ I, S\ \mathtt{)} \wedge A \leq_{Int} B\end{array} \tag{3}$$

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

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

$\mathbb{K}$ [1] 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 [6] specifications, which can then be used for executing programs and for analysing them. $\mathbb{K}$ also offers some support for symbolic calculus, including a connection to the Z3 SMT solver [7]. We have used these components in a prototype tool implementing our deductive system for program equivalence.

The prototype is a $\mathbb{K}$ definition that is currently able to "execute" equivalence formulas for the IMP language. Such formulas are programs of another language, say, IMPeq, thus, our prototype is a $\mathbb{K}$ definition of the IMPeq language. We start by showing how the prototype proves the equivalence formula from Example 11.

The prototype is given the following IMPeq program as input:

```
<k> L : for i from symAExp(a1) to symAExp(a2) do symStmt(s) </k>
<env> i -> symInt(i1), SymMap(M) </env>
~
<k> L: i := symAExp(a1);
       while (i <= symAExp(a2)) do (
         symStmt(s);
    L:   i := i + 1 )
</k>
<env> i -> symInt(i1), SymMap(M) </env>
if  not occurs(i, symStmt(s))
using observers: allObs;
```

The expressions `symAExp(_)`, `symStmt(_)`, `symInt(_)`, `SymMap(_)` denote symbolic variables of sort `AExp`, `Stmt`, `Int`, `Map`, respectively. The syntax used by the prototype for formulas is close to their mathematical notation. One difference is the introduction of labels, to be explained later. Another difference is the introduction of a notation for observers (denoting the observation relation) in the syntactical construct `using observers:`. The special constant `allObs` means that the two programs have the same variables and all of them are observers.

The above input is loaded into a configuration of the form:

$$\langle\langle\langle\langle P_1\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C\rangle_{\mathsf{cond}}\rangle_{\mathsf{goal}}\rangle_{\mathsf{goals}}\langle\rangle_{\mathsf{hypos}}\langle\texttt{allObs}\rangle_{\mathsf{obs}}.$$

The equivalence formula is stored into a `goal` cell, contained in the `goals` cell that stores all goals generated during the formula's execution (i.e., the sets G). The cell `hypos` stores the hypotheses F generated during execution; at the beginning it is empty.

The inference rules of the proof system are the semantical rules of the IMPeq language. We first focus on the [Derive] rule, which is the "heart" of the proof system. For goals $g \triangleq \langle\langle P_1\rangle_{\mathsf{k}}\langle E_1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \sim \langle\langle P_2\rangle_{\mathsf{k}}\langle E_2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *if* C having exactly one derivative, [Derive] is implemented by two rules, which have the form:

$$\begin{array}{l}\langle\langle P_1\rangle_{\mathsf{k}}\langle E_1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2\rangle_{\mathsf{k}}\langle E_2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C\rangle_{\mathsf{cond}} \wedge g \notin (g \wedge \Omega) \Rightarrow \\ \quad \langle\langle P_1'\rangle_{\mathsf{k}}\langle E_1'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2'\rangle_{\mathsf{k}}\langle E_2'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C'\rangle_{\mathsf{cond}} \qquad (4)\\ \langle\langle\langle\langle P_1\rangle_{\mathsf{k}}\langle E_1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2\rangle_{\mathsf{k}}\langle E_2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C\rangle_{\mathsf{cond}}\rangle_{\mathsf{goal}}\rangle_{\mathsf{goals}}\langle F\rangle_{\mathsf{hypos}} \wedge g \in (g \wedge \Omega) \Rightarrow \\ \quad \langle\langle\langle\langle P_1'\rangle_{\mathsf{k}}\langle E_1'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2'\rangle_{\mathsf{k}}\langle E_2'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C'\rangle_{\mathsf{cond}}\rangle_{\mathsf{goal}}\rangle_{\mathsf{goals}}\langle F \cup \{g\}\rangle_{\mathsf{hypos}} \qquad (5)\end{array}$$

These rules implement two particular, yet important cases of [Derive]: either an empty set of hypotheses is added ($F' = \emptyset$) or the goal itself is added to the hypotheses ($F' = \{g\}$). We note that derivatives are directly computed using the semantical rules of IMP, based on the observation made earlier in the paper that, for IMP (as well as in many pratical cases) most general unifiers are computed by matching with the rules of the language. If we omit the check of the condition $g \notin (g \wedge \Omega)$, the rules (4) are automatically computed from the semantical rules by $\mathbb{K}$'s *configuration abstraction* mechanism.

In order to reduce the number of generated hypotheses and implicitly the calls of the rule (5), we have added labels to statements with the following role: the rule (5) only matches applied when the first statements of P_1 and of P_2 have the same label. The membership $g \in (g \wedge \Omega)$ is only checked in these cases.

Most goals only have one derivative and are handled by the rules (4,5). The exception is the case where the first statement of P_1 or P_2 has the form `if` B `then` S_1 `else` S_2 when B is a symbolic Boolean variable (i.e., not the constants *true* or *false*): it is then possible that both $C \wedge B$ and $C \wedge \neg B$ be satisfiable, in which case both rules for `if` are applicable. This is handled by a rule of the form

$$\begin{array}{l}\langle\langle\langle P_1\rangle_{\mathsf{k}}\langle E_1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2\rangle_{\mathsf{k}}\langle E_2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C\rangle_{\mathsf{cond}}\rangle_{\mathsf{goal}} \wedge (sat(C \wedge B) \wedge sat(C \wedge \neg B)) \Rightarrow \\ \quad \langle\langle\langle\langle P_1'\rangle_{\mathsf{k}}\langle E_1'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2'\rangle_{\mathsf{k}}\langle E_2'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C \wedge B\rangle_{\mathsf{cond}}\rangle_{\mathsf{cfg}}\rangle_{\mathsf{goal}} \\ \quad \langle\langle\langle\langle P_1''\rangle_{\mathsf{k}}\langle E_1''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle\langle P_2''\rangle_{\mathsf{k}}\langle E_2''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\langle C \wedge \neg B\rangle_{\mathsf{cond}}\rangle_{\mathsf{cfg}}\rangle_{\mathsf{goal}} \qquad (6)\end{array}$$

where the satisfiability predicate $sat()$ is evaluated by calling the Z3 SMT solver. The other significant rule of the proof system: [Reduce], is implemented by:

$$\langle \cdots \langle g \rangle_{\mathsf{goal}} \cdots \rangle_{\mathsf{goals}} \langle F \rangle_{\mathsf{hypos}} \wedge F \vdash g \Rightarrow \langle \cdots \mathit{success} \cdots \rangle_{\mathsf{goals}} \langle F \rangle_{\mathsf{hypos}} \tag{7}$$

where $F \vdash g$ if either both programs in g are empty and the environments are in the observation relation or g is an instance of a formula $f \in F$, and *success* is a special, idempotent constant that we use to denote a proved goal.

An equivalence formula g holds if a configuration of the form

$$\langle \mathit{success} \rangle_{\mathsf{goals}} \langle \cdots \rangle_{\mathsf{hypos}} \langle \cdots \rangle_{\mathsf{obs}}$$

is reachable from the initial configuration generated by g. Assuming that the formula that expresses the equivalence between `for` and `while` is included in the file `for-while.peq`, the command that tries to proves this reachability is:

```
$ krun for-while.peq --search
         --pattern "=>* < T > < goals > success </ goals > B:Bag </ T >"
         --bound "1"
Search results:
Solution 1, state 1020:
B:Bag -->
 <hypos>
 ...
```

where the value of the `pattern` option gives the pattern of the configuration is searched for in zero, one or more steps, and the `bound` value gives the maximum number of solutions after the search process is finished (in our case one is enough). The cell `hypos` contains the hypotheses gathered during execution; it denotes the weak bisimulation witnessing the equivalence of `for` and `while`.

To conclude this section we show another example of equivalent programs, which both have infinite executions:

```
<k>                                    <k>
  L : while (true) do (                  L : while (true) do (
        s := s + i * i;                        s := s + k;
        i := i + 1                             k := k + i + i + 1;
      )                          ∼             i := i + 1
</k>                                         )
<env>  i -> symInt(i01),                 </k>
       SymMap(M1)                        <env> i -> symInt(i02),
</env>                                         k -> symInt(k0), SymMap(M2)
                                         </env>
if symInt(k0) == symInt(i02)*symInt(i02) and symInt(i01) == symInt(i02)
using observers: n, s;
```

The programs compute infinitely many sums of squares of integer numbers in two different ways. The observation relation is given by the values of `n` and `s`. Despite the infinite executions of the programs our proof system terminates after a few steps (actually, in fewer steps than if we had placed a symbolic or even a large bound on the number of loop iterations, because the proof system never needs to test against a bound, thus, all generated goals only have one derivative).

Using other logics supported by $\mathbb{K}$ Framework (e.g., matching logic [8]), this prototype can be transformed into a powerful prover for program equivalence.

9 Conclusion, Related Work, 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. This also gives us the possibility of capturing standard equivalence relations, e.g., "two programs are equivalent if, when presented with the same input, if they terminate they produce the same output".

The approach works also for symbolic programs, in which some expressions and statements are symbolic variables, denoting sets of concrete programs obtained by replacing the symbolic variables by concrete expressions/statements.

Related Work The earliest reference to program equivalence we know of is [9], and other early ones are [10,11]. An extension of the program-equivalence problem: designing a compiler ensuring the equivalence of its source and target programs, was declared a grand challenge in computer science by Tony Hoare [12]. An exhaustive bibliography on the program-equivalence topic is outside the scope of this work. We classify some contributions according to the following criteria:

General vs. language or program-specific. Most contributions target specific languages (e.g., C and the code produced from it by compilation [13,14]), or classes of languages (functional [15], microcode [16], CLP [17]). Some target particular kinds of programs: for example, the equivalence of programs that differ from each other in few respects in [2], the equivalence of recursive procedures in [3].

Interactive vs. automatic. Most contributions, including the early ones cited above, remained essentially theoretical due to the absence of mature tools to support them. Among the mechanised ones, some are based on interactive theorem provers [13] whereas others are fully automatic; e.g. the CADP toolset [18] automatically checks for various equivalences on a process algebra-like language.

Industrial vs. academic Few approaches deal with real language and industrial-size programs in those languages. Among the closest to such high standards are [13,14,16,18]. This is in contrast to equivalence checking in hardware, which has entered mainstream industrial practice (see, e.g., [19] for a survey on this).

Our proof system is inspired by *circular coinduction* [4], whose proof system allows one to prove equalities of data structures such as infinite streams and regular expressions. A notable difference between the present approach and [4] is that our specifications are rewrite theories (meant to define the semantics of programming languages), whereas those of [4] are behavioural equational theories, a special class of equational specifications with visible and hidden sorts.

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.

We are also planning to combine our program-equivalence verification with matching logic [20] verification. Matching logic is an automatic, language-independent formal verification framework for languages with a rewrite-based semantics. In matching logic one annotates statements with formulas that can be seen as pre-post conditions and invariants; not surprisingly, it works best for structured languages (like our source language mentioned above). The idea is to prove matching logic properties on programs in the source language, and guarantee, via the compiler's correctness (proved by symbolic program equivalence) that the compiled programs in the target language satisfy those properties as well.

Another promising approach is to merge the proof systems of program equivalence and of matching logic into one uniform proof system that would allow one to prove equivalence properties and matching logic properties simultaneously.

References

1. G. Roşu and T.-F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
2. Ofer Strichman. Regression verification: Proving the equivalence of similar programs. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, page 63. Springer, 2009.
3. Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. In Zohar Manna and Doron Peled, editors, *Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2010.
4. Grigore Roşu and Dorel Lucanu. Circular coinduction – a proof theoretical foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.
5. 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.
6. 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.
7. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on*

Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

8. 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.
9. Yuri Yanov. Logical operator schemes. *Kibernetika*, 1, 1958.
10. Guy Cousineau and Patrice Enjalbert. Program equivalence and provability. In Jirí Becvár, editor, *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 237–245. Springer, 1979.
11. J.A. Bergstra and J.W. Klop. *Formal Description of Programming Concepts - II*, chapter Formal proof systems for program equivalence, pages 289–302. North Holland Publishing Company, 1983.
12. Tony Hoare. The verifying compiler: A grand challenge for computing research. In GÃűrel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 262–272. Springer Berlin / Heidelberg, 2003.
13. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
14. George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *PLDI*, pages 83–94. ACM, 2000.
15. 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.
16. 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.
17. 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.
18. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.
19. Fabio Somenzi and Andreas Kuehlmann. *Electronic Design Automation For Integrated Circuits Handbook*, volume 2, chapter 4: Equivalence Checking. Taylor & Francis, 2006.
20. Grigore Rosu and Andrei Stefanescu. Towards a unified theory of operational and axiomatic semantics. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, volume 7392 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2012.

A 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, $(\gamma_1, \gamma_2) \in [\![g]\!]$. □

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

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

Proposition 1. *Let* $t \in T_\Sigma(Var)$, $\sigma, \sigma' : Var \to T_\Sigma(Var)$, $\rho, \rho', \rho'' : Var \to \mathcal{T}$, $V \in Var$. *Then*

1. *if* $var(t) \subseteq V$, *then* $t(\rho|_V \cup \rho'|_{\overline{V}}) = t\rho|_V$;
2. *if* $var(t) \subseteq V$, *then* $t(\rho|_V \cup \rho'|_{\overline{V}}) = t(\rho|_V \cup \rho''|_{\overline{V}})$;
3. *if* $dom(\sigma) \cap dom(\sigma') = \emptyset$, $ran(\sigma) \subseteq V$ *and* $var(t) \subseteq dom(\sigma)$ *then* $t(\sigma \cup \sigma')(\rho|_V \cup \rho'|_{\overline{V}}) = t\sigma\rho|_V$.

Proof. By structural induction on t. The acs of interest is that when t is a variable. We consider here only the third affirmation. If t is a variable x, then $x \in dom(\sigma)$ is defined and $x\sigma \in T_\Sigma(V)$ and hence $x\sigma\rho|_V$ is defined.

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* $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ *and* $\gamma_1 \neq \gamma_1'$ *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 $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ and $\gamma_1 \neq \gamma_1'$, by the definition of $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ 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 $g' \triangleq (r\sigma \sim \pi_2 \ \mathit{if}\ (C \wedge b)\sigma \wedge \bigwedge \sigma) \in \Delta^l(g)$. Since:

$$
\begin{aligned}
\gamma_1' &= r\rho' && \text{(by the definition of } \gamma_1') \\
&= r(\rho'|_{\overline{var(l)}} \cup \rho'|_{var(l)}) \\
&= r(\rho|_{\overline{var(l)}} \cup \rho'|_{var(l)}) && (\rho' \text{ chosen s.t. } \rho'|_{\overline{var(l)}} = \rho|_{\overline{var(l)}} \text{ allowed by Rem. 1}) \\
&= r\sigma\eta && \text{(by the def. of } \rho'', \sigma \text{ and } \eta) \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \sigma\eta|_{var(\pi_2)}) \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) && (\sigma\eta|_{var(\pi_2)} = \rho|_{var(\pi_2)} \text{ by def. of } \rho'', \sigma \text{ and } \eta) \\
&= r(\sigma\eta|_{\overline{var(\pi_2)}} \cup \sigma\rho|_{var(\pi_2)}) && (\sigma \text{ is the identity on } var(\pi_2)) \\
&= r\sigma(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) \\
&= (r\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) && (8) \\
\gamma_2 &= \pi_2\rho && \text{(by the definition of } \gamma_2) \\
&= \pi_2(\rho|_{var(\pi_2)} \cup \rho|_{\overline{var(\pi_2)}}) \\
&= \pi_2(\rho|_{var(\pi_2)} \cup \eta|_{\overline{var(\pi_2)}}) && (\text{since valuations on } \overline{var(\pi_2)} \text{ do not affect } \pi_2) \\
&= \pi_2(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) && (9)
\end{aligned}
$$

and

$$
\begin{aligned}
((C \wedge b)\sigma)(\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) &= (C \wedge b)(\sigma\eta|_{\overline{var(\pi_2)}} \cup \rho|_{var(\pi_2)}) \\
&= (C \wedge b)\sigma\eta \\
&= (C \wedge b)\rho'' \\
&= \mathit{true} && (10)
\end{aligned}
$$

and

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

$$= \left(\bigwedge_{x \in var(l)} x\rho' = x\sigma\eta \right) \wedge \left(\bigwedge_{y \in var(\pi_1)} y\rho = y\sigma\eta \right) \quad (x\eta = x\rho', y\eta = y\rho)$$

$$= \left(\bigwedge_{x \in var(l)} x\rho'' = x\rho'' \right) \wedge \left(\bigwedge_{y \in var(\pi_1)} y\rho'' = y\rho'' \right) \quad \text{(def. of. } \rho'')$$

$$= true \tag{11}$$

it follows that (8)-(11) imply $(\gamma_1', \gamma_2) \in (\!|g'|\!)$ using the valuation $\eta \cup \rho|_{var(\pi_2)}$. □

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

Proof. Let $(\gamma_1, \gamma_2) \in [\![g]\!]$, $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ and $\gamma_1 \neq \gamma_1'$. Since $[\![g]\!] \subseteq (\!|g|\!)$, by Lemma 2 there exists $g' \in \Delta^l(g)$ such that $(\gamma_1', \gamma_2) \in (\!|g'|\!)$. But since $(\gamma_1, \gamma_2) \in [\![g]\!]$, γ_1 and γ_2 are the origins infinite computations, and (by determinism) so is γ_1'. Thus, $(\gamma_1', \gamma_2) \in [\![g']\!]$. □

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

Proof. We prove the first statement; the second one is proved similarly. By the definition of $[\![f]\!]$ and the determinism of $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ there exists exactly one transition $\gamma_1 \rightarrow \gamma_1'$. We show first that $\gamma_1 \neq \gamma_1'$. Assume $\gamma_1 = \gamma_1'$, thus, γ_1 is a final configuration, which means that the pattern π_1 contains an empty program. This contradicts $\Delta^l(f) \neq \emptyset$. Thus, $\gamma_1 \neq \gamma_1'$, and Corollary 1 concludes the proof. □

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 $\rho : Var \rightarrow \mathcal{T}$ 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 $\eta : Var \rightarrow \mathcal{T}$ 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 $\eta' : Var \rightarrow \mathcal{T}$ 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 $\rho_i : Var \rightarrow \mathcal{T}$ 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

$$\pi_i\sigma_i\rho = \pi_i\sigma_i\rho_i|_{ran(\sigma_i)} = \pi_i\sigma_i\rho_i = \pi_i(\eta|_{var(f)} \cup \eta'|_{\overline{var(f)}}) = \gamma_i. \tag{12}$$

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,

$$C(\sigma_1 \cup \sigma_2)\rho = C\eta|_{var(f)} = \mathit{true} \tag{13}$$

In a similar way we get

$$C'(\sigma_1 \cup \sigma_2)\rho = C'\eta|_{var(g)} = \mathit{true} \tag{14}$$

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 $i = 1, 2$:

$$\begin{aligned}
(\bigwedge \sigma_i)\rho &= (\bigwedge_{x \in dom(\sigma_i)} x = x\sigma_i)\rho \\
&= \bigwedge_{x \in dom(\sigma_i)} x\rho = x\sigma_i\rho \\
&= \left(\bigwedge_{x \in var(\pi_i)} x\rho = x\sigma_i\rho\right) \wedge \left(\bigwedge_{x \in var(\pi_i')} x\rho = x\sigma_i\rho\right) \\
&= \left(\bigwedge_{x \in var(\pi_i)} x\eta = x\sigma_i\rho_i\right) \wedge \left(\bigwedge_{x \in var(\pi_i')} x\eta' = x\sigma_i\rho_i\right) \\
&= \mathit{true}
\end{aligned} \tag{15}$$

From (12), (13), (14), and (15) we obtain $(\gamma_1, \gamma_2) \in (\!|h|\!) \subseteq (\!|f \wedge g|\!)$. □

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 there are infinite executions starting in γ_1, γ_2. □

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 $\gamma_1 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1''$ and $\gamma_2 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2''$.

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. $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ 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 $\gamma_1' \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1''$ and $\gamma_2 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2''$. Hence $\gamma_1 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1''$ and $\gamma_2 \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2''$, 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 $\mathcal{S} \nvdash f$, $[\![f]\!] \subseteq [\![\Omega]\!]$ 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 $(\gamma_1, \gamma_2) \in \bigcup_{f \in \mathcal{F}, \mathcal{S} \nvdash} [\![f]\!]$. 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 $(\gamma_1, \gamma_2) \in \bigcup_{f \in \mathcal{F}, \mathcal{S} \nvdash} [\![f]\!]$, there is $f \in \mathcal{F}$ with $\mathcal{S} \nvdash f$ 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 (cf. Section 7.2) 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* $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ *and* $\gamma_1 \neq \gamma_1'$.

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 $\rho'' : Var \to \mathcal{T}$ such that:

- $\gamma_1' = r\sigma\rho''$
- $\gamma_2' = \pi_2\rho''$
- $(C \wedge C')\sigma\rho'' = true$.

Let $\gamma_1 = l\sigma\rho''$. Then, $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1' = r\sigma\rho''$ by applying the rule $l \wedge C' \Rightarrow r$ with valuation $\sigma\rho''$ on γ_1 (note that $C'\sigma\rho'' = true$). Since $\gamma_1 \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_1'$ was obtained by applying a rule, $\gamma_1 \neq \gamma_1'$. Moreover, $\sigma \in U(\pi_1, l)$, which means $l\sigma = \pi_1\sigma$ and implies $l\sigma\rho'' = \pi_1\sigma\rho''$. Since $C\sigma\rho'' = true$, and $\gamma_2' = \pi_2\rho'' = \pi_2\sigma\rho''$ (the last equality because $dom(\sigma) = var(\pi_1) \cup var(l)$, whose intersection with $var(\pi_2)$ 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 $g : \pi_1 \sim \pi_2$ *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 $\gamma_1' \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_1$ and $\gamma_2' \Rightarrow_{\mathcal{S}}^{*\mathcal{T}} \gamma_2$. 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 $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$. □