Program Equivalence by Circular Reasoning

Dorel Lucanu, Vlad Rusu

Program Equivalence by Circular Reasoning

Dorel Lucanu, Vlad Rusu

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

RESEARCH REPORT
N° 8116
2013
Project-Team Dreampal

Program Equivalence by Circular Reasoning

Dorel Lucanu*, Vlad Rusu†

Project-Team Dreampal

Research Report n° 8116 — 2013 — 33 pages

Abstract: We propose a logic and a deductive system for stating and automatically proving the equivalence of programs written in deterministic languages having a rewriting-based operational semantics. The chosen equivalence is parametric in a so-called observation relation, and it says that two programs satisfying the observation relation will inevitably be, in the future, in the observation relation again. This notion of equivalence generalises several well-known equivalences, and is shown to be appropriate for deterministic programs. 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 given program-equivalence problem. We show that our approach is suitable for proving equivalence for terminating and non-terminating programs as well as for concrete and symbolic programs. The latter are programs in which some statements or expressions are symbolic variables. By proving the equivalence between symbolic programs, one proves the equivalence of (infinitely) many concrete programs obtained by replacing the variables by concrete statements or expressions. The approach is illustrated by proving program equivalence in two languages from different programming paradigms. The examples in the paper, as well as other examples, can be checked using an online tool.

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

* Al. I. Cuza University of Iaşi, Romania, dlucanu@info.uaic.ro

† Inria Lille Nord-Europe, France, Vlad.Rusu@inria.fr

RESEARCH CENTRE
LILLE – NORD EUROPE

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

Résumé : Nous proposons une logique et un système déductif pour exprimer et prouver automatiquement l'équivalence de programmes dans des langages déterministes munis de sémantiques opérationnelles définies par réécriture. Le système déductif proposé est de nature circulaire; nous démontrons qu'il est correct et faiblement complet. Ces deux résultats signifient que, lorsqu'il termine, notre système résout correctement le problème d'équivalence de programmes tels que nous l'avons posé. Nous montrons que ce système fonctionne autant pour des programmes qui terminent que pour des programmes qui ne terminent pas. Les programmes dits symboliques, dans lesquels certaines expressions ou instructions restent non-interprétés, peuvent également être traités par notre approche. La démonstration d'une équivalence entre deux programmes symboliques revient à démontrer l'équivalence entre une infinité potentielle de programmes concrets, qui sont des instances des programmes symboliques obtenues en remplaçant les variables symboliques par des instructions ou des expressions concrètes. L'approche est illustrée par la preuve d'équivalence de programmes dans deux langages appartenant à des paradigmes de programmation différents. Les exemples contenus dans l'article, ainsi que d'autres exemples, peuvent être essayés dans un outil en ligne.

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

1 Introduction

In this paper we propose a formal notion of program equivalence, together with a language-independent logic for expressing this notion and a deductive system for automatically proving it. Programs can belong to any deterministic language whose semantics is specified by a set of rewrite rules. The equivalence we consider is parametric in a certain observation relation, and it requires that, for all programs satisfying the observation relation, their executions eventually lead them into satisfying the observation relation again. The proof system is circular: its conclusions can be re-used as hypotheses in a controlled way. Since the problem it tries to solve is undecidable, our proof system is not guaranteed to terminate. When it does terminate, it solves the program-equivalence problem as stated, thanks to its soundness and weak completeness properties.

The proposed framework is shown suitable for terminating and nonterminating programs as well as for concrete and for *symbolic programs.* The latter are programs in which some expressions and/or statements are *symbolic variables*, which denote sets of concrete programs obtained by substituting the symbolic variables by concrete expressions and/or statements. Thus, by proving the equivalence between symbolic programs, one proves in just one shot the equivalence of (possibly, infinitely) many concrete programs.

Example 1.1 *We want to translate general programs with* `for`*-loops into programs with* `while`*-loops. This amounts to translating the symbolic program in the left-hand side to the one in the right-hand side.*

`for` I `from` A `to` B `do{` S `}` $\qquad$ 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 expressions (A, B), and program statement (S). We assume that the `for`*-loop and* `while`*-loop statements have independent semantics (i.e., the* `for` *instruction is not desugared into to a* `while` *instruction) and the* `for` *loop does not modify the counter I, nor any program variables occurring in A, B (note that program variables are identifiers). If we prove the equivalence between these two symbolic programs then we also prove that every concrete instance of the* `for`*-loop is equivalent to its translation to the corresponding* `while`*-loop.*

Example 1.2 *The second example illustrates the equivalence of non-terminating corecursive programs. Such programs are similar to recursive programs, but their terminating condition is missing, and therefore they describe non-terminating computations. Here we consider corecursive programs over infinite sequences of integers (also called* streams*). Such a program is expressed using a set of equations; for each equation, the left-hand side is the name of a function being defined, possibly with parameters, and the right-hand side is the function's body. Let us consider the corecursive program consisting of the following equations:*

`hd(` x `:` xs `)` $\approx$ x `;`	`tl(` x `:` xs `)` $\approx$ xs `;`
`zero` $\approx$ `0 : zero;`	`one` $\approx$ `0 : one;`
`blink` $\approx$ `0 : 1 : blink;`	`zip(` xs `,` ys `)` $\approx$ `hd(` xs `) : zip(` ys `, tl(` xs `));`

where x ranges over integers and xs over streams. Obviously, the complete evaluation of `zero` *produces the infinite sequence $0 : 0 : 0 : \ldots$, and the evaluation of* `one` *produces the infinite sequence $1 : 1 : 1 : \ldots$,* `blink` *produces the infinite sequence $0 : 1 : 0 : 1 : \ldots$, and* `zip(` xs, ys `)` *produces a stream that alternates the elements of the two streams given to it as parameters. The function* `hd(` xs `)` *returns the first element of the stream xs (this is the only function in the languge that does not produce a stream), and* `tl(` xs `)` *returns the stream obtained from xs after removing the first element. A well-known equivalence over streams is that of* `blink` *and* *`zip(zero,one)` and many proofs of it can be found in the literature. We use this example to show that our notion of equivalence is general enough for being applicable to terminating programs as well as to non-terminating ones. The example also serves to illustrate the language-genericity of our approach.*

Hereafter we often refer to symbolic programs just as "programs". A typical use of our framework consists in:

1. formally defining a programming language $\mathcal{L}$, whose concrete programs are ground terms over a certain signature defining the language's syntax, and whose symbolic programs are terms with variables over that signature. The operational semantics of $\mathcal{L}$ is assumed given as a conditional term-rewriting system;

2. automatically constructing a new language definition $\mathcal{L} \times \mathcal{L}$, whose programs are pairs of programs of $\mathcal{L}$[1];

3. applying our deductive system to programs in $\mathcal{L} \times \mathcal{L}$.

Running the deductive system amounts essentially to symbolically executing the semantics of $\mathcal{L} \times \mathcal{L}$, which consists in applying the rewrite rules in the semantics with *unification* instead of matching; details are given in the paper. This may lead to one of the following outcomes:

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

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

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

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

Contributions A language-independent logic and a proof system suitable for stating and proving the equivalence of concrete and of symbolic programs as well as of terminating and non-terminating ones. Programs can be written in any deterministic language that has a formal operational semantics based on term rewriting. We prove the soundness and weak completeness of the proof system, which ensure that the system correctly solves the program equivalence problem as stated. The approach is illustrated on two different languages. The examples in the paper, as well as and other examples, can be tried using an online tool, currently available at `http://fmse.info.uaic.ro/tools/K/?tree=examples/prog-equiv/README`.

With respect to the conference paper [1]: the equivalence relation is reformulated in terms of a Linear Temporal Logic (LTL) formula, and the soundness/weak completeness proofs are simpler, thanks to an encoding of executions of our proof system as the building of proofs for the

[1] We have developped the approach for the equivalence of programs belonging to one language $\mathcal{L}$ for simplicity reasons. However, considering two languages $\mathcal{L}$ and $\mathcal{L}'$ poses no conceptual difficulty and can even be reduced to the one-language case. Indeed, any program in language $\mathcal{L}$ and any program in language $\mathcal{L}'$ are also programs in the language $\mathcal{L} \uplus \mathcal{L}'$, i.e, in the disjoint union of the two languages. This union obtained by (possibly) renaming some common language constructions to avoid ambiguity.

LTL formulas in question. The genericity of the approach is illustrated by considering programs in two different programming paradigms.

We also generalise (for the needs of the program-equivalence approach) a generic symbolic execution technique introduced in [2]: by executing semantical rules with unification instead of matching we also allow, e.g., the symbolic execution of symbolic statements in addition to the symbolic data considered in [2].

Related Work An exhaustive bibliography on the program-equivalence problem is outside the scope of this paper, as this problem is even older than the program-verification problem. Among the recent works perhaps the closest to ours is [3]. They also deal with the equivalence of parameterised programs (symbolic, in our terminology) and define equivalence in terms of bisimulation. Their approach is, however, very different from ours. One major difference lies in the models of programs: [3] use CFGs (control flow graphs) of programs, while we use the operational semantics of languages. CFGs are more restricted, e.g., they are not well adapted to recursive or object-oriented programs, whereas operational semantics do not have these limitations. Of course, our advantage will only become apparent when we actually apply our approach to such programs.

Other closely related recent works are [4, 5, 6]. The first one targets programs that include recursive procedures, the second one exploits similarities between single-threaded programs in order to prove their equivalence, and the third one extends the latter to multi-threaded programs. They use operational semantics (of a specific language, which focuses on recursive procedure definition) and proof systems, and formally prove their proof system's soundness. In [4] they make a useful classification of equivalence relations used in program-equivalence research, and use these relations in their work.

However, all the relations classified in [4] are of an input/output nature: for given (sequences of) inputs, programs generate equal (sequences of) outputs and/or do not terminate. Such relations are well adapted for concrete programs with inputs and outputs, but not to symbolic programs with symbolic statements, for which a clear input-output relation may not exist. Indeed, symbolic statements may denote arbitrary concrete statements - including ones that do not perform input/output - actually, when symbolic programs are concerned, one cannot even rely on the existence of inputs and outputs. One may rely, however, on the observations of the effects of symbolic statements on the program's environment (e.g., values of variables). Our notion of equivalence (parameterised by a certain observation relation) allows this, both for finitely and for infinitely many repeated observations. Moreover, we also show that some of the relations from [4] can be encoded in our relation by adding information to the program environment.

Many works on program equivalence arise from the verification of compilation in a broad sense. At one end there is full compiler verification [7], and at the other end, the so-called translation validation, i.e., the individual verification of each compilation [8] (we only cite two of the most relevant recent works). As also observed by [3], symbolic program verification can also be used for certain compilers, in which one proves the equivalence of each basic instruction pattern from the source language with its translation in the target language. The application of this observation to the verification of a compiler (from another project we are involved in) is ongoing and will be presented in another paper.

Several other works have targeted specific classes of languages: functional [9], microcode [10], CLP [11]. In order to be less language-specific some works advocate the use of intermediate languages, such as [12], which works on the Boogie intermediate language. Only a few approaches, among which [7, 10], deal with real-life language and industrial-size programs in those languages. This is in contrast to the equivalence checking of hardware circuits, which has entered mainstream industrial practice (see, e.g., [13] for a survey).

Our proof system is inspired by that of *circular coinduction* [14], which allows one to prove equalities of data structures such as infinite streams and regular expressions. A notable difference between the present approach and [14] is that our specifications are essentially rewrite systems (meant to define the semantics of programming languages), whereas those of [14] are behavioural equational theories, a special class of equational specifications with visible and hidden sorts.

Symbolic linear temporal-logic model checking in term-rewriting systems, which we here use for proving program equivalence, was earlier studied in [15]. There are differences in expressiveness: we only use certain specific LTL formulas for encoding equivalence, whereas [15] handle full LTL; on the other hand, they consider unconditional term-rewriting systems only, whereas we also consider conditional term-rewriting systems. For our approach, which is based on programming-language semantics, having conditional rewriting systems is essential since unconditional rules are not expressive enough to express nontrivial languages semantics. There are also differences in the underlying deduction mechanisms: [15] rely on powerful unification-modulo-theories algorithms, while our unification algorithm delegates deduction to satisfiability modulo theory (SMT) solvers.

Organisation. After this introduction, Section 2.1 presents our running examples: IMP, a simple imperative language, and STREAM, a corecursive language for handling streams of integers. Both languages are defined in $\mathbb{K}$ [16], a formal framework for defining operational semantics of programming languages. Our approach is, however, independent of the $\mathbb{K}$ framework and the IMP language; hence, we present a general, abstract mechanism for language definitions in Section 3, and show how $\mathbb{K}$ definitions are instances of that mechanism. In Section 4 we define a unification operation and prove some properties about it, which are used in Section 5 where we present a generic *symbolic execution* approach for languages defined in the proposed mechanism. We formally relate symbolic execution to concrete execution, which we use later in the paper for proving the correctness properties (soundness and weak completeness) of our proof system.

In Section 6 we recap linear-temporal logic (LTL). This is then used in Section 7, which contains our proposed definition for program equivalence as the satisfiaction of certain LTL formulas over an execution of the transition system generated by (concretely) executing a pair of programs. The formula says that the programs will repeatedly satisfy a certain *observation relation*; this relation is a parameter of the approach. The syntax and semantics of a logic capturing the chosen equivalence are defined.

The proof system for proving equivalence formulas is presented in Section 8, together with its soundness and weak completeness. The properties say that, when it terminates, the proof system correctly answers to the question of whether its input (which is a set of formulas of program-equivalence logic) denotes equivalent programs. Their proofs are based on building proof witnesses for LTL formulas expressing equivalence. The witnesses are obtained by symbolically executing the pair of programs under investigation.

In Section 9 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 also defined in $\mathbb{K}$. Finally, the conclusion and future work are presented in Section 10.

Acknowledgments This work was partially supported by Contract 161/15.06.2010, SMISC-SNR 602-12516.

2 Two Examples of Programming Languages and their Semantics in $\mathbb{K}$

We use two different languages as running examples: IMP, a simple imperative language, and STREAM, a language for manipulating integer streams. We present their formal definitions in the $\mathbb{K}$ framework [16], a formal environment for defining programming languages, type systems, and analysis tools. The main ingredients of a $\mathbb{K}$ definition are *computations*, *configuration*, and *rules*. Computations are sequences of elementary computational tasks, which consist of e.g. adding two numbers, or transforming the program being executed. A configuration is a nested structure of *cells* that include all the data structures required for executing a program. The rules describe how the configurations are modified when the computational tasks are performed. For details on the theoretical background of $\mathbb{K}$ readers can consult [16].

$\mathbb{K}$ language definitions can be executed and analysed using tools from the $\mathbb{K}$ environment. Examples of language definitions and related analysis tools can be found on the web page `http://kframework.org`.

2.1 IMP - A Simple Imperative Language

The first 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 [16]. The syntax of IMP is described in Figure 1 and is mostly self-explained. The attribute (given as an annotation) *strict* from the syntax means the arguments of the annotated expression/statement are evaluated before the expression/statement itself is evaluated/executed. If the attribute has as arguments a list of natural numbers, then only the arguments in positions specified by the list are evaluated before the expression/statement. The *strict* attribute is actually syntactic sugar for a set of $\mathbb{K}$ rules, briefly presented later in the section. The *configuration* of an IMP program consists of code to be executed and an enviroment mapping identifiers to integers. In $\mathbb{K}$, this is written as a nested structure of *cells*: here, a top cell `cfg`, having a cell `k` containing code and a cell `env` (see Figure 3). The sort *Code*[2] contains statements and arithmetic and Boolean expressions. The empty code is denoted by $\cdot$, and code sequencing is denoted by $\curvearrowright$. Note that this is different from the sequencing operation `;` 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 elementary expressions. An example of sequence of computations is given Figure 3b); this sequence is obtained by applying the *heating* rules generated by the strict attribute for the statement `if` and the operator `<`. The heating/cooling rules are explained latter. 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., $\mathtt{x} \mapsto 3$.

The semantics of IMP is given by a set of rules (see Figure 2) 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, negation, and `if` statement, the semantics of each operator and statement is described by exactly one rule.

In Figure 2, the operations $\mathit{lookup} : \mathit{Map} \times \mathit{Id} \to \mathit{Int}$ and $\mathit{update} : \mathit{Map} \times \mathit{Id} \times \mathit{Int} \to \mathit{Map}$ are part of the domain of maps and have the usual meanings: *lookup* returns the value of an identifier in a map, and *update* modifies the map by adding (or, if it exists, by updating) the binding of an identifier to a value.

[2] In the $\mathbb{K}$ terminology the sort *Code* is called *K*. We changed its name in order to avoid confusions due to name overloading.

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

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

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

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

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

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

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

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

a) $\mathbb{K}$ Configuration of IMP

$$\left\langle \begin{array}{l} \langle \texttt{x} \curvearrowright \square \texttt{ < 0} \curvearrowright \texttt{if (} \square \texttt{) y = 0; else y = 1;}\rangle_{\mathsf{k}} \\ \langle \texttt{x} \mapsto 3 \quad \texttt{y} \mapsto -7\rangle_{\mathsf{env}} \end{array} \right\rangle_{\mathsf{cfg}}$$

b) An IMP configuration snapshot

Figure 3: IMP configurations.

In addition to the rules in Figure 2 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 *heating/cooling* rules (automatically generated by $\mathbb{K}$):

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

where BE ranges over $BExp\backslash\{false, true\}$, B ranges over $\{false, true\}$, and $\square$ is a special variable destined to receive the value of BE once it is computed. Finally, the following rules, related to the construction of terms of sort *Code*, complete de $\mathbb{K}$ definition of IMP: for all $C, C_1, C_2, C_3 : Code$: $\langle\langle \cdot \curvearrowright C\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle C\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$, and $\langle\langle (C_1 \curvearrowright C_2) \curvearrowright C_3\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle C_1 \curvearrowright (C_2 \curvearrowright C_3)\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$. The first one says that the empty sequence $\cdot$ is a left-neutral element, and the second one says that $\curvearrowright$ is right-associative.

2.2 STREAM - a Simple Language for Corecursive Programs

Corecursive programs differ from recursive ones by the fact that their termination condition is missing. Besides functional languages, which typically use corecursion for handling infinite data structures, several other languages have been extended to support such features (see, e.g., [17] for an extension of Prolog, and [18] for an extension of Java). An example of a corecursive program was given in Section 1. Here we present a simple language for writing such programs over integer streams (= infinite sequence of integers). The standard semantics for corecursive functions is based on lazy evaluation, which delays the evaluation of expressions until their value is needed. For infinite expressions this evaluation is always partial, in the sense that only a finite part of the infinite expression is evaluated, e.g., a finite prefix of an infinite stream.

Therefore, we say that a stream expression is a *result value* if it is of the form $i : SE$, where the integer i is the first element of the stream and SE is the rest of the stream expression. Beside the constructor `_:_`, two functions, often called destructors, are essential in handling streams: `hd(` xs `)`, which returns the first element of the stream, and `tl(` xs `)`, which returns the stream obtained after the first element is removed.

The syntax of the STREAM language is given in Figure 4. There are three expression kinds: *BEXp* - for boolean expressions, *IEXp* - integer expressions, and *SExp* - stream expressions. The operator $X \triangleleft B \triangleright Y$ is the conditional operator `if` B `then` X `else` Y written in a Hoare-like syntax, which is more compact. There are two kinds of statements (specifications *SSpec*): integer function specifications, written as f `:=` ... or $f(\ldots)$ `:=` ... (these are usual, recursive functions), and stream specifications, written as $s \approx \ldots$ or $s(\ldots) \approx \ldots$ (these are corecursive functions)

A STREAM program is a sequence of function specifications, followed by an expression to be evaluated. The $\mathbb{K}$ configuration for STREAM programs is represented in Figure 5. As the snapshot suggests, the cell specs stores definitions of recursive and corecursive functions. The right-hand side of a function definition is a λ-expression, defined as follows:

$$\begin{array}{rl} BExp ::= & Bool \\ & \mid IExp \texttt{ = } IExp \text{ [strict]} \\ & \mid BExp \texttt{ \& } BExp \text{ [strict(1)]} \\ & \mid \texttt{!}\ BExp \text{ [strict]} \\ IExp ::= & Int \\ & \mid \texttt{hd (} SExp \texttt{)} \text{ [strict]} \\ & \mid IExp \texttt{ + } IExp \text{ [strict]} \\ & \mid IExp \triangleleft BExp \triangleright SExp \text{ [strict(2)]} \end{array} \quad \begin{array}{rl} SExp ::= & Id \\ & \mid \texttt{tl (} SExp \texttt{)} \text{ [strict]} \\ & \mid Id \texttt{ (} SExps \texttt{)} \\ & \mid IExp \texttt{ : } SExp \\ Exp ::= & IExp \mid SExp \end{array} \quad \begin{array}{rl} SSpec ::= & Id \texttt{ := } SExp \texttt{ ;} \\ & \mid Id \texttt{ (} Ids \texttt{) := } IExp \texttt{ ;} \\ & \mid Id \texttt{ (} Ids \texttt{)} \approx SExp \texttt{ ;} \\ SPgm ::= & SSpecs\ Exp \\ SExps ::= & List\{SExp, ","\} \\ Ids ::= & List\{Id, ","\} \\ SSpecs ::= & List\{SSpec, ""\} \end{array}$$

$$Code ::= IExp \mid SExp \mid Exp \mid SSpec \mid SSpecs \mid SPgm \mid Code \curvearrowright Code$$

Figure 4: $\mathbb{K}$ Syntax of STREAM

$$\text{Val} ::= \lambda \texttt{ (} Ids \texttt{) . } SExp$$

The cell out includes results of evaluations, which can be integers or stream result values, depending on the type of the expression to be evaluated. This cell is essential for the stream equivalence definition since it defines their observational relation. Note that the evaluation of stream expressions is an nonterminating process and the out cell includes only finite approximations of streams.

The $\mathbb{K}$ semantics of STREAM is given in Figure 6. The rules giving semantics for the boolean/integer operators that are similar to those from the IMP definition and are omitted. The semantics of a function call expression consists of replacing the expression with the function body, where the formal parameters are replaced by the actual arguments (if any). The other rules are self-explained.

Example 2.1 *We illustrate the semantics of* STREAM *on the following example. Assume that the current configuration is* $\langle\langle \texttt{tl(one)}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$. *In order to evaluate the expression* $\texttt{tl(one)}$, *the above configuration is* heated *to*

$$\langle\langle \texttt{one}\curvearrowright\texttt{tl(}\,\square\,\texttt{)}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

Now, the rule evaluating stream functions without parameters (the ninth one in Figure 6) is applied and generates the term $\langle\langle \texttt{1:one}\curvearrowright\texttt{tl(}\,\square\,\texttt{)}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$. *The expression* $\texttt{1:one}$ *is a result value and the corresponding cooling rule is applied, producing*

$$\langle\langle \texttt{tl(1:one)}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

Applying the rule for $\texttt{tl}$ *and then the rule for function calls we obtain*

$$\langle\langle \texttt{1:one}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

Since the content of the k cell consists only of $\texttt{1:one}$, *the rule writing in the* out *cell (the first one in Figure 6) can be applied the following configuration is obtained:*

$$\langle\langle \texttt{one}\rangle_{\mathsf{k}}\langle \texttt{one} \mapsto \lambda().\ \texttt{1:one}\rangle_{\mathsf{specs}}\langle \texttt{[]:1}\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

This sequence of rules can be repeated arbitrarily (but finitely) many times, and they generate arbitrarily larger (but finite) approximations of the infinite stream one *in the* out *cell.*

$$\langle\langle Code\rangle_{\mathsf{k}}\langle Map\rangle_{\mathsf{sspecs}}\langle SExp\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

a) $\mathbb{K}$ configuration of STREAM

$$\left\langle \begin{array}{c} \langle \texttt{tl(blink)}\curvearrowright\texttt{hd(}\;\square\;\texttt{)}\rangle_{\mathsf{k}} \\ \langle \texttt{zero} \mapsto \lambda().\,\texttt{0 : zero one} \mapsto \lambda().\,\texttt{1 : one blink} \mapsto \lambda().\,\texttt{0 : 1 : blink}\rangle_{\mathsf{specs}} \\ \langle \texttt{1 : 0}\rangle_{\mathsf{out}} \end{array} \right\rangle_{\mathsf{cfg}}$$

b) A STREAM configuration snapshot

Figure 5: STREAM configurations.

$\langle\langle I\ \texttt{:}\ SE\rangle_{\mathsf{k}}\langle OE\rangle_{\mathsf{out}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle SE\rangle_{\mathsf{k}}\langle OE\ \texttt{:}\ I\rangle_{\mathsf{out}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I\rangle_{\mathsf{k}}\langle OE\rangle_{\mathsf{out}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle\rangle_{\mathsf{k}}\langle OE\ \texttt{:}\ I\rangle_{\mathsf{out}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I_1\ \texttt{=}\ I_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \land I_1 =_{Int} I_2 \Rightarrow \langle\langle true\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle I_1\ \texttt{=}\ I_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \land I_1 \neq_{Int} I_2 \Rightarrow \langle\langle false\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle \texttt{hd(}\ I\ \texttt{:}\ \_\ \texttt{)}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle IE_1 \lhd true \rhd IE_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle IE_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle IE_1 \lhd false \rhd IE_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle IE_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle \texttt{tl(}\ \_\ \texttt{:}\ SE\ \texttt{)}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle SE\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle F\ \cdots\rangle_{\mathsf{k}}\langle\cdots\ F \mapsto \lambda().SE\ \cdots\rangle_{\mathsf{specs}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle SE\ \cdots\rangle_{\mathsf{k}}\langle\cdots\ F \mapsto \lambda().SE\ \cdots\rangle_{\mathsf{specs}}\ \cdots\rangle_{\mathsf{cfg}}$

$\langle\langle F(Vs)\ \cdots\rangle_{\mathsf{k}}\langle\cdots\ F \mapsto \lambda(Xs).SE\ \cdots\rangle_{\mathsf{specs}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle SE[Vs/Xs]\ \cdots\rangle_{\mathsf{k}}\langle\cdots\ F \mapsto \lambda(Xs).SE\ \cdots\rangle_{\mathsf{specs}}\ \cdots\rangle_{\mathsf{cfg}}$

where the operation $[\_/\_]$ denotes syntactical substitution.

Figure 6: $\mathbb{K}$ Semantics of STREAM.

3 Language Definitions

Our program-equivalence approach is independent of the formal framework used for defining languages as well as from the languages being defined. We thus propose a general notion of language definition and illustrate it later in the section on the $\mathbb{K}$ definition of IMP. We assume the reader is familiar with the basics of algebraic specification and rewriting. A language $\mathcal{L}$ is defined by:

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for configurations and a subsignature Σ^{Bool} for Booleans with their usual constants and operations. Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all data sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{Data}$, and that terms of sort *Cfg* have subterms denoting statements (which are programs in the syntax of $\mathcal{L}$) remaining to be executed. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(Var)$ 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. For terms $t_1,\ldots,t_n$ we let $var(t_1,\ldots,t_n) \triangleq var(t_1) \cup \cdots var(t_n)$. For any *substitution* $\sigma : Var \to T_\Sigma(Var)$ and term $t \in T_\Sigma(Var)$ we denote by $t\sigma$ the term obtained by applying the substitution σ to t. We

use the *diagrammatical order* for the composition of substitutions, i.e., for substitutions σ and σ', the composition $\sigma\sigma'$ consists in first applying σ then σ'.

2. A Σ-algebra $\mathcal{T}$, over which the semantics of the language is defined. $\mathcal{T}$ interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data}-algebra $\mathcal{D}$. $\mathcal{T}$ interprets non-data sorts as ground terms over the signature of the form $(\Sigma \setminus \Sigma^{Data}) \cup \mathcal{D}$ (1) i.e., the elements of $\mathcal{D}$ are added to the signature $\Sigma \setminus \Sigma^{Data}$ as constants of their respective sorts. That is, a language is parametric in the way its data are implemented; it just assumes there is such an implementation $\mathcal{D}$. This is important for technical reasons (existence of a unique most general unifier, discussed below). Let $\mathcal{T}_s$ denote the elements of $\mathcal{T}$ that have the sort s; the elements of $\mathcal{T}_{Cfg}$ are called *configurations*. Any *valuation* $\rho : Var \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(Var) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. If $b \in T_{\Sigma,Bool}(Var)$ then we write $\rho \models b$ iff $b\rho = \mathcal{D}_{true}$, where $b\rho$ is the Boolean value obtained by applying ρ to b. For simplicity, we often write *true*, *false* instead of $\mathcal{D}_{true}, \mathcal{D}_{false}$.

3. A set $\mathcal{S}$ of rewrite rules $l \wedge b \Rightarrow r$, whose formal definition is given later in the section.

We explain these concepts on the IMP example. Each nonterminal from the syntax (*Int*, *Bool*,...) is a sort in Σ. Each production from the syntax defines an operation in Σ; for instance, the production $AExp ::= AExp + AExp$ defines the operation $\_+\_ : AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the configuration sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle X = I \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{Cfg}(Var)$, where X is a variable of sort *Id*, I is a variable of sort *Int*, C is a variable of sort *Code* (the rest of the computation), and *Env* is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets *Int* as the set of integers, the operations like $+_{Int}$ (cf. Figure 2) as the corresponding usual operation on integers, *Bool* as the set of Boolean values $\{false, true\}$, the operation like $\wedge_{Bool}$ as the usual Boolean operations, the sort $Map_{Id,Int}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers *Id* and I over the integers *Int*. The fact that maps are modified only by the *update* operation ensures that each identifier is bound to at most one integer value. The other sorts, *AExp*, *BExp*, *Stmt*, and *Code*, are interpreted in the algebra $\mathcal{T}$ as ground terms over a modification of the form (1) of the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{Int} 0$ `then skip else skip` is interpreted in $\mathcal{T}$ as `if` *true* `then skip else skip`, since $\mathcal{D}$ interprets $1 >_{Int} 0$ as $\mathcal{D}_{true}(= true)$.

The rewrite rules describe the transitions over configurations, whose formal definition is given below.

Definition 3.1 (pattern [19]) *A* pattern *is an expression of the form $\pi \wedge \phi$, where $\pi \in T_{\Sigma,Cfg}(Var)$ is a* basic pattern *and $\phi \in T_{\Sigma,Bool}(Var)$ is a boolean term called the pattern's* condition. *If $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ we write $(\gamma, \rho) \models \pi \wedge \phi$ for $\gamma = \pi\rho$ and $\rho \models \phi$. We let $[\![\pi \wedge \phi]\!]$ denote the set $\{\gamma \mid$ there exists ρ such that $(\gamma, \rho) \models \pi \wedge \phi\}$.*

For any set of patterns Φ we let $[\![\Phi]\!] \triangleq \bigcup_{\varphi\in\Phi}[\![\varphi]\!]$. A basic pattern π thus defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy. In [19] patterns are encoded as FOL formulas, hence the conjunction notation $\pi \wedge b$. In this paper we keep the notation but separate basic patterns from constraining formulas. We often identify basic patterns π with patterns $\pi \wedge true$.

Examples of patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$. An example of configuration that satisfies the second pattern is $\langle\langle(4 \;/\; 3) \curvearrowright \mathtt{skip}\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto 5\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$.

Remark 3.1 *Any pattern* $\pi \wedge \phi$ *can be transformed into a "semantically equivalent" pattern* $\pi' \wedge \phi'$ *(i.e.., $[\![\pi \wedge \phi]\!] = [\![\pi' \wedge \phi']\!]$) such that π' is linear and all its data subterms are variables. For this, just replace all duplicated variables and all non-variable data subterms of π by fresh variables, and add constraints to equate in ϕ the fresh variables to what they replaced. The transformations are presented in detail in [20].*

Example 3.1 *The pattern* $\langle\langle X \;/\; Y\rangle_{\mathsf{k}}\langle Y \mapsto A +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A \neq_{Int} -1$ *with X, Y variables of sort Id and A of sort Int is nonlinear because Y occurs twice. Moreover, it contains the non-variable data terms $A +_{Int} 1$. It is thus transformed into the pattern*

$$\langle\langle X \;/\; Y\rangle_{\mathsf{k}}\langle Y' \mapsto A'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge Y' =_{Id} Y \wedge_{Bool} A' =_{Int} A +_{Int} 1 \wedge_{Bool} A \neq_{Int} -1$$

The proof system we propose in Section 8 uses as a basic block the testing of inclusions of the form $[\![\varphi]\!] \subseteq [\![\varphi']\!]$. Therefore we need criteria for such inclusions. The following lemmas define sufficient conditions.

Proposition 3.1 *Let π' and π be two basic patterns and σ a substitution such that $\pi'\sigma = \pi$, $y\sigma = y$ for all $y \notin var(\pi')$, and $var(\pi') \cap var(\pi \wedge \phi) = \emptyset$. Then $[\![\pi \wedge \phi]\!] = [\![\pi' \wedge (\bigwedge_\sigma \wedge \phi)]\!]$, where $\bigwedge_\sigma$ denotes the conjunction $\bigwedge_{x \in var(\pi')} x = x\sigma$.*

Proof We prove the equality of the two sets by double inclusion.

($\subseteq$) Let $\gamma \in [\![\pi \wedge \phi]\!]$. Then there is $\rho : Var \to \mathcal{T}$ such that $\gamma = \pi\rho$ and $\rho \models \phi$. Let ρ' denote the valuation $\rho' : Var \to \mathcal{T}$ given by $x\rho' = x\sigma\rho$ for $x \in var(\pi')$, and $y\rho' = y\rho$ for $y \notin var(\pi')$. We have $\pi'\rho' = \pi'\sigma\rho = \pi\rho = \gamma$. Since $var(\phi) \cap var(\pi') = \emptyset$, it follows that $\rho' \models \phi$ iff $\rho \models \phi$. Finally, $var(\pi') \cap var(\pi) = \emptyset$ implies $var(\pi') \cap var(\sigma(x)) = \emptyset$ and hence $x\sigma\rho' = x\sigma\rho$ for $x \in var(\pi')$. Now, $\pi'\rho' = \gamma$ that implies $\gamma \in [\![\pi' \wedge \bigwedge_\sigma \wedge \phi)]\!]$. Since γ was chosen arbitrarily, it follows that $[\![\pi \wedge \phi]\!] \subseteq [\![\pi' \wedge (\bigwedge_\sigma \wedge \phi)]\!]$.

($\supseteq$) Assume that $\gamma \in [\![(\pi' \wedge \bigwedge_\sigma \wedge \phi)]\!]$. Then there is $\rho' : Var \to \mathcal{T}$ such that $\gamma = \pi'\rho'$, and $\rho' \models (\bigwedge_\sigma \wedge \phi)$. From $\rho' \models \bigwedge_\sigma$ we get $x\rho' = x\sigma\rho'$ for $x \in var(\pi')$, which implies $\gamma = \pi'\rho' = \pi'\sigma\rho = \pi\rho$. Hence $\gamma \in [\![\pi \wedge \phi]\!]$. Since γ was chosen arbitrarily, it follows that $[\![\pi' \wedge (\bigwedge_\sigma \wedge \phi)]\!] \subseteq [\![\pi \wedge \phi]\!]$. ☐

Proposition 3.2 *Let $\pi \wedge \phi$ and $\pi' \wedge \phi'$ be two patterns and σ a substitution such that $\pi'\sigma = \pi$, $y\sigma = y$ for all $y \notin var(\pi')$, and $var(\pi') \cap var(\pi \wedge \phi) = \emptyset$. If $\bigwedge_\sigma \wedge \phi$ implies ϕ', then $[\![\pi \wedge \phi]\!] \subseteq [\![\pi' \wedge \phi']\!]$.*

Proof Let $\gamma \in \pi \wedge \phi$. Then there is a valuation ρ such that $\gamma = \pi\rho$ and $\rho \models \phi$. Let ρ' be defined as in Proposition 3.1. Then $\pi'\rho' = \gamma$ and $\rho' \models \bigwedge_\sigma \wedge \phi$ that implies $\rho' \models \phi'$ by the hypotheses. Hence $\gamma \in [\![\pi \wedge \phi]\!]$. Since γ was chosen arbitrarily, we may conclude the conclusion of the proposition. ☐

Remark 3.2 *The conditions $var(\pi') \cap var(\pi \wedge \phi) = \emptyset$ and $y\sigma = y$ for all $y \notin var(\pi')$ required by Proposition 3.1 and Proposition 3.2 can be easily obtained by a variable renaming.*

Proposition 3.3 *Let $\pi \wedge \phi$ and $\pi' \wedge \phi'$ two patterns such that there is a substitution σ with $(\pi' \wedge \phi')\sigma = \pi \wedge \phi$. Then $[\![\pi \wedge \phi]\!] \subseteq [\![\pi' \wedge \phi']\!]$.*

Proof Let $\gamma \in \pi \wedge \phi$. Then there is a valuation ρ such that $\gamma = \pi\rho$ and $\rho \models \phi$. Let ρ' be defined by $x\rho' = x\sigma\rho$ for each x in Var. It follows $\pi'\rho' = \pi'\sigma\rho = \pi\rho = \gamma$ and similarly $\phi'\rho' = \phi\rho$ that implies $\rho' \models \phi'$. Since γ was chosen arbitrarily, we may conclude the conclusion of the proposition. ☐ We are now ready to define semantical rules and the transition system that they generate.

Definition 3.2 (semantical rule and transition system [19]) *A* rule *is a pair of patterns of the form $l \wedge b \Rightarrow r$ (where r is the pattern $r \wedge true$). Any set $\mathcal{S}$ of rules defines a labelled transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$, where $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ iff there are $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$.*

Assumption 1 *We assume without restriction of generality that for all rules $l \wedge b \Rightarrow r \in \mathcal{S}$, l is linear and all its data subterms are variables. The generality is not restricted because the pattern $l \wedge b$ in the rule $l \wedge b \Rightarrow r \in \mathcal{S}$ can always be replaced by an equivalent one (cf. Remark 3.1) with the desired properties. This transformation of rules does not modify the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$.*

4 Unification

We shall be using unification both for defining our program-equivalence proof system and for proving properties about it. In this section we define unification and prove a technical lemma used later in the paper.

Definition 4.1 (Unifiers) *A* symbolic unifier *of two terms t_1, t_2 is any substitution $\sigma : var(t_1) \uplus var(t_2) \to T_\Sigma(Z)$ for some set Z of variables such that $t_1\sigma = t_2\sigma$. A* concrete unifier *of terms t_1, t_2 is any valuation $\rho : var(t_1) \uplus var(t_2) \to \mathcal{T}$ such that $t_1\rho = t_2\rho$. A symbolic unifier σ of two terms t_1, t_2 is a* most general unifier of t_1, t_2 with respect to concrete unification *whenever, for all concrete unifiers ρ of t_1 and t_2, there is a valuation η such that $\sigma\eta = \rho$. We often call a symbolic unifier satisfying the above a* most general unifier[3].

We say that terms t_1, t_2 are symbolically (resp. concretely) unifiable if they have a symbolic (resp. concrete) unifier. The next lemma gives conditions under which concretely unifiable terms are symbolically unifiable.

Lemma 4.1 *All linear, concretely unifiable terms $t_1, t_2 \in T_\Sigma(Var)$, such that all their data subterms are variables, are symbolically unifiable by a most general unifier $\sigma^{t_1}_{t_2} : var(t_1) \uplus var(t_2) \to T_\Sigma(var(t_1) \uplus var(t_2))$.*

Proof By induction on the structure of, say, t_1. In the base case, $t_1 \in Var$, and we take $\sigma^{t_1}_{t_2} \triangleq (t_1 \mapsto t_2) \uplus Id|_{var(t_2)}$, i.e., $\sigma^{t_1}_{t_2}$ maps t_1 to t_2, and is the identity on $var(t_2)$. Obviously, $\sigma^{t_1}_{t_2}$ is a unifier of t_1, t_2, since $t_1\sigma^{t_1}_{t_2} = t_2$. To show that $\sigma^{t_1}_{t_2}$ is most general, consider any concrete unifier of t_1, t_2, say, ρ. Then, $t_1\sigma^{t_1}_{t_2}\rho = t_2\rho$ because $\sigma^{t_1}_{t_2}$ maps t_1 to t_2, and $t_2\rho = t_1\rho$ because ρ is a concrete unifier. Thus, $t_1\sigma^{t_1}_{t_2}\rho = t_1\rho$. Moreover, for all $x \in var(t_2)$, $x\sigma^{t_1}_{t_2}\rho = x\rho$ since $\sigma^{t_1}_{t_2}$ is the identity on $var(t_2)$. Thus, for all $y \in var(t_1) \uplus var(t_2) (= \{t_1\} \uplus var(t_2))$, $y\sigma^{t_1}_{t_2}\rho = y\rho$, which proves the fact that $\sigma^{t_1}_{t_2}$ is a most general unifier (by taking $\eta = \rho$ in Definition 4.1 of unifiers). The fact that the codomain of $\sigma^{t_1}_{t_2}$ is $T_\Sigma(var(t_1) \uplus var(t_2))$ results from its construction.

In the inductive step, $t_1 = f(s_1, \ldots, s_n)$ with $f \in \Sigma \setminus \Sigma^{Data}$ [4] $n \geq 0$, and $s_1, \ldots, s_n \in T_\Sigma(Var)$. For t_2 there are two subcases:

- t_2 is a variable. Then, let $\sigma^{t_1}_{t_2} \triangleq (t_2 \mapsto t_1) \uplus Id|_{var(t_1)}$, i.e., $\sigma^{t_1}_{t_2}$ maps t_2 to t_1, and is the identity on $var(t_1)$. We prove that $\sigma^{t_1}_{t_2}$ is a most general unifier with codomain $T_\Sigma(var(t_1) \uplus var(t_2))$ like in the base case.

[3] even though the standard notion of most general unifier in algebraic specifications and rewriting is a different one.

[4] $f \in \Sigma \setminus \Sigma^{Data}$ because the contrary would mean that t_1 has a *Data* sort, in contradiction with the lemma's hypotheses.

- $t_2 = g(u_1, \ldots, u_m)$ with $g \in \Sigma$, $m \geq 0$, and $u_1, \ldots, u_m \in T_\Sigma(Var)$. Let ρ be a concrete unifier of t_1, t_2, thus, $(f\rho)(s_1\rho \ldots s_n\rho) =_{\mathcal{T}} (g\rho)(u_1\rho \ldots u_m\rho)$, where we emphasize by subscripting the equality symbol with $\mathcal{T}$ that the equality is that of the model $\mathcal{T}$. Since $\mathcal{T}$ interprets non-data terms as ground terms over the modified signature (1), we have $f\rho = f$, which implies $f = g$, $g\rho = g$, $m = n$, and $s_i\rho = u_i\rho$ for $i = 1, \ldots, n$. Since t_1 and t_2 are linear and all their data subterms are variables, the subterms s_i and u_i also have these properties. Using the induction hypothesis we build most-general-unifiers $\sigma^{s_i}_{u_i}$ of s_i and u_i, which have codomains $T_\Sigma(var(s_i) \uplus var(u_i))$, for $i = 1, \ldots, n$. Let then $\sigma^{t_1}_{t_2} \triangleq \biguplus_{i=1}^{n} \sigma^{s_i}_{u_i}$. First, $\sigma^{t_1}_{t_2}$ is a substitution of $var(t_1) \uplus var(t_2)$ into $T_\Sigma(var(t_1) \uplus var(t_2))$ since $var(t_1) = \uplus_{i=1}^{n} var(s_i)$ and $var(t_2) = \uplus_{i=1}^{n} var(u_i)$. Note that these equalities hold thanks to the linearity of t_1, t_2. Second, $\sigma^{t_1}_{t_2}$ is a unifier of t_1, t_2 since all $\sigma^{s_i}_{u_i}$ are so. Third, we prove that $\sigma^{t_1}_{t_2}$ is a most general unifier of t_1, t_2. Consider any concrete unifier ρ of t_1 and t_2, thus, $s_i\rho = u_i\rho$ for $i = 1, \ldots, n$. From the fact that all the $\sigma^{s_i}_{u_i}$ are most-general-unifiers of s_i and u_i for $i = 1, \ldots, n$, we obtain the existence of valuations η_i such that $\sigma^{s_i}_{u_i}\eta_i = \rho|_{(var(s_i) \uplus var(u_i))}$, for $i = 1, \ldots, n$. Then, $\eta \triangleq \biguplus_{i=1}^{n} \eta_i$, which is also well-defined thanks to the linearity of t_1 and t_2, has the property that $\sigma^{t_1}_{t_2}\eta = \rho$, which proves that $\sigma^{t_1}_{t_2}$ is a most general unifier of t_1 and t_2 and concludes the proof.

□

5 Symbolic Execution

In this section we present a symbolic execution approach for languages defined using the language-definition framework presented in the previous section. We prove that the transition system generated by symbolic execution forward-simulates the one generated by concrete execution, and that the transition system generated by concrete execution backward-simulates the one generated by symbolic execution (restricted to satisfiable patterns). This is used later for proving correctness results for our program-equivalance deduction system.

Symbolic execution consists of applying the semantical rules over patterns using most general unifiers. This generalises the symbolic execution approach proposed in [2], where unification was encoded using matching with modified rules, and which did not allow for symbolic statements. Symbolic execution generates a *symbolic transition system* whose states are patterns, and whose transition relation is obtained by applying rewrite rules with the most-general unifiers whose construction is given by Lemma 4.1.

Definition 5.1 (Symbolic transition relation) *$\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ iff $\varphi \triangleq \pi \wedge \phi$, there is a rule $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ with $var(l) \cap var(\pi) = \emptyset$ and such that l, π are concretely unifiable, and $\varphi' = r\sigma^l_\pi \wedge (\phi \wedge b)\sigma^l_\pi$, where σ^l_π is unique, most general symbolic unifier of l, π constructed as in the proof of Lemma 4.1.*

Definition 5.2 *The* derivative *of a pattern is the set of patterns that can be obtained by one symbolic execution step: $\Delta_{\mathcal{S}}(\varphi) \triangleq \{\varphi' \mid \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'\}$. A pattern φ is* derivable *if $\Delta_{\mathcal{S}}(\varphi)$ is a nonempty set.*

In the rest of the paper, for patterns $\varphi \triangleq \pi \wedge \phi$ we let $var(\varphi) \triangleq var(\pi, \phi)$, and for rules $\alpha \triangleq l \wedge r \Rightarrow b$ we let $var(\alpha) \triangleq var(l, b, r)$. Moreover, for symbolic transitions $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ we assume without restriction on generality that $var(\varphi) \cap var(\alpha) = \emptyset$, which can always be obtained by variable renaming. We also omit to write the subscript $\mathcal{S}$ in the derivatives notation whenever it is understood from the context.

Lemma 5.1 *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$ then there exists φ' such that $\gamma' \in [\![\varphi']\!]$ and $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$.*

Proof Let $\varphi \triangleq \pi \wedge \phi$. From $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ we obtain the rule $\alpha \triangleq l \wedge r \Rightarrow b$ and the valuation $\rho : Var \to \mathcal{T}$ such that $\gamma = l\rho$, $b\rho = true$, and $\gamma' = r\rho$. From $\gamma \in [\![\varphi]\!]$ we obtain the valuation $\mu : Var \to \mathcal{T}$ such that $\gamma = \pi\mu$ and $\phi\mu = true$. Thus, l and π are concretely unifiable (by their concrete unifier $\rho|_{var(l)} \uplus \mu|_{var(\pi)}$). Using Lemma 4.1 we obtain their unique most-general symbolic unifier σ^l_π, whose codomain is $T_\Sigma(var(l) \uplus var(\pi))$. Let then $\eta : var(l) \uplus var(\pi) \to \mathcal{T}$ be the valuation such that $\sigma^l_\pi \eta = \rho|_{var(l)} \uplus \mu|_{var(\pi)}$. We extend σ^l_π to $var(\varphi, \alpha)$ by letting it be the identity on $var(\varphi, \alpha) \setminus var(l, \pi)$, and extend η to $var(\varphi, \alpha)$ such that $\eta|_{var(b,r)\setminus var(l)} = \rho|_{var(b,r)\setminus var(l)}$ and $\eta|_{var(\phi)\setminus var(\pi)} = \mu|_{var(\phi)\setminus var(\pi)}$. With these extensions we have $x(\sigma^l_\pi \eta) = x(\rho \uplus \mu)$ for all $x \in var(\varphi, \alpha)$.

Let $\varphi' \triangleq r\sigma^l_\pi \wedge (\phi \wedge b)\sigma^l_\pi$: we have the transition $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ by definition of the symbolic transition system. There remains to prove $\gamma' \in [\![\varphi']\!]$.

- on the one hand, $(r\sigma^l_\pi)\eta = r(\sigma^l_\pi \eta) = r(\rho \uplus \mu) = r\rho = \gamma'$; thus, $(\gamma', \eta) \models r\sigma^l_\pi$;
- on the other hand, $((\phi \wedge b)\sigma^l_\pi)\eta = \phi(\sigma^l_\pi \eta) \wedge b(\sigma^l_\pi \eta) = \phi(\rho \uplus \mu) \wedge b(\rho \uplus \mu) = \phi\mu \wedge b\rho = true$ since $\phi\mu = b\rho = true$; thus; $\eta \models ((\phi \wedge b)\sigma^l_\pi)$.

The two above items imply $(\gamma', \eta) \models r\sigma^l_\pi \wedge (\phi \wedge b)\sigma^l_\pi$, i.e., $(\gamma', \eta) \models \varphi'$, which concludes the proof. □

Corollary 5.1 *For every concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ there is a symbolic execution $\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_n \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for $i = 0, 1, \ldots$.*

Lemma 5.2 *If $\gamma' \in [\![\varphi']\!]$ and $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ then there exists $\gamma \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$.*

Proof From $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq l \wedge r \Rightarrow b$ we obtain $\varphi' = r\sigma^l_\pi \wedge (\phi \wedge b)\sigma^l_\pi$. From $\gamma' \in [\![\varphi']\!]$ we obtain $\eta : Var \to \mathcal{T}$ such that $\gamma' = (r\sigma^l_\pi)\eta$ and $((\phi \wedge b)\sigma^l_\pi)\eta = true$. We extend σ^l_π to $var(\varphi, \alpha)$ by letting it be the identity on $var(\varphi, \alpha) \setminus var(l, \pi)$. Let $\rho : Var \to \mathcal{T}$ be defined by $x\rho = x(\sigma^l_\pi \eta)$ for all $x \in var(\varphi, l)$, and $x\rho = x\eta$ for all $x \in Var \setminus var(\varphi, l)$, and let $\gamma \triangleq l\rho$. From $\gamma' = (r\sigma^l_\pi)\eta$ and the definition of ρ we obtain $\gamma' = r\rho$. From $((\phi \wedge b)\sigma^l_\pi)\eta = true$ we obtain $b(\sigma^l_\pi \eta) = true$, i.e., $b\rho = true$, which together with $\gamma \triangleq l\rho$ and $\gamma' = r\rho$ gives $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. There remains to prove $\gamma \in [\![\varphi]\!]$.

- From $\gamma = l\rho$ using the definition of ρ we get $\gamma = l\rho = l(\sigma^l_\pi \eta) = (l\sigma^l_\pi)\eta = (\pi\sigma^l_\pi)\eta = \pi(\sigma^l_\pi \eta) = \pi\rho$;
- From $((\phi \wedge b)\sigma^l_\pi)\eta = true$ we obtain $(\phi\sigma^l_\pi)\eta = \phi(\sigma^l_\pi \eta) = \phi\rho = true$.

Since $\varphi \triangleq \pi \wedge \phi$, the last two items imply $(\gamma, \rho) \models \varphi$, which completes the proof. □ We call a symbolic execution *feasible* if all its patterns are satisfiable (a pattern φ is satisfiable if there is a configuration γ such that $\gamma \in [\![\varphi]\!]$).

Corollary 5.2 *For every feasible symbolic execution $\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_1 \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_n \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ there is a concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for $i = 0, 1, \ldots$.*

6 Linear Temporal Logic

The notion of program equivalence we propose uses Linear Temporal Logic (LTL) formulas, defined below.

A *Kripke structure* is a tuple $(S, \Rightarrow, P, \lambda, S^i)$ where S is a set of *states*, $\Rightarrow \subseteq S \times S$ is a (total) *transition relation*, P is a set of *propositions*, $\lambda : S \to 2^P$ is the *labelling function*, and $S^i \subseteq S$ is the set of *initial states.*

An *execution* $e \triangleq s_0, \ldots, s_n, \ldots$ is a sequence of states such that $s_0 \in S^i$ and $s_j \Rightarrow s_{j+1}$ for all $j \in \mathbb{N}$. The *suffix* of an execution e from $l \in \mathbb{N}$, denoted by e^l, is the sequence such that $e^l(j) = s_{l+j}$ for all $j \in \mathbb{N}$.

LTL formulas are generated by the grammar $\psi ::= \mathit{true} \mid p \mid \bigcirc\psi \mid \psi \wedge \psi \mid \neg\psi \mid \psi\,\mathcal{U}\,\psi$ for all $p \in P$. Standard abbreviations are $\mathit{false} \triangleq \neg\mathit{true}$, $\psi_1 \vee \psi_2 \triangleq \neg(\neg\psi_1 \wedge \neg\psi_2)$, $\Diamond\psi \triangleq \mathit{true}\,\mathcal{U}\psi$, and $\Box\psi \triangleq \neg\Diamond\neg\psi$.

Given an execution $e \triangleq s_0, \ldots, s_n, \ldots$ of a Kripke structure $(S, \Rightarrow, P, \lambda, S^i)$ and an LTL formula ψ, the satisfaction relation $e \models \psi$ is inductively defined over the structure of ψ as follows:

- $e \models \mathit{true}$;
- $e \models p$ iff $p \in \lambda(s_0)$;
- $e \models \bigcirc p$ iff $e^1 \models p$
- $e \models \psi_1 \wedge \psi_2$ if $e \models \psi$ and $e \models \psi_2$;
- $e \models \neg\psi$ iff it is not the case that $e \models \psi$;
- $e \models \psi_1\,\mathcal{U}\psi_2$ iff there exists $k > 0$ such that $e^k \models \psi_2$ and for all $0 \leq j < k$, $e^j \models \psi_1$.

We will be be interested in formulas of the form $\Box\Diamond p$. Using the semantics of LTL, an execution $e = s_0, \ldots, s_n, \ldots$ satisfies $\Box\Diamond p$ iff it has an infinite subsequence $s_{i_1}, \ldots s_{i_m}, \ldots$ such that $p \in \lambda(s_{i_j})$ for all $j \in \mathbb{N}$.

7 Defining Program Equivalence

We define in this section our notion of program equivalence and a logic for stating equivalence properties.

Assumption 2 *We assume without restriction of generality that the transition system* $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}}$ *) has no terminal states (i.e., its transition relation is total). This can always be obtained by adding to* $\mathcal{S}$ *rules of the form* $\pi \Rightarrow \pi$ *for all non-derivable patterns* π*, which just add self-loops to terminal states of* $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$.

Remark 7.1 *Strictly speaking, it is not programs that are the subject of equivalence, but full configurations (of which programs are just one component). Indeed, program executions depend on the rest of the configuration (e.g., initial values of the variables, ...). Hence, the equivalence relation is a relation on* $\mathcal{T}_{\mathit{Cfg}}$.

We consider a given *observation relation* $O \subseteq \mathcal{T}_{\mathit{Cfg}} \times \mathcal{T}_{\mathit{Cfg}}$, which shall serve as a parameter to our equivalence. Then, we say that two configurations are *observationally equivalent* if they are in the observation relation.

Observational equivalence should be understood as a purely local property of configuration pairs, such as, e.g., a given set of variables have the same values in both configurations. Then, our notion of program equivalence requires that starting from any two observationally equivalent configurations, by executing the programs in the configuration one will eventually encounter observationally equivalent configurations again.

This expressed by the LTL formula $O \wedge \Box\Diamond O$, which captures precisely the informal meaning given above. In order to formalise this observation, it will be convenient to consider, for a given language definition $\mathcal{L}$, the language definition denoted by $\mathcal{L}^2$, whose configurations are pairs of configurations of $\mathcal{L}$ and whose rewrite rules are those of $\mathcal{L}$, lifted at the level of configurations of $\mathcal{L}^2$; that is, each semantical rule $l \wedge b \Rightarrow r$ of $\mathcal{L}$ generates two rules of $\mathcal{L}^2$: $\langle l, X\rangle \wedge b \Rightarrow \langle r, X\rangle$ and $\langle Y, l\rangle \wedge b \Rightarrow \langle Y, r\rangle$ where $\langle \_, \_\rangle$ is the configuration constructor for $\mathcal{L}^2$ and X, Y are variables of sort *Cfg* for $\mathcal{L}$ that do not occur in the rest of the rule.

Let $\mathcal{S}_l^2$ denote the set of rules of $\mathcal{L}^2$ of the form $\langle l, X\rangle \wedge b \Rightarrow \langle r, X\rangle$, and $\mathcal{S}_r^2$ denote the set of rules of $\mathcal{L}^2$ of the form $\langle Y, l\rangle \wedge b \Rightarrow \langle Y, r\rangle$. We denote by $\mathcal{S}^2$ the whole set of rules of $\mathcal{L}^2$, i.e., $\mathcal{S}^2 = \mathcal{S}_l^2 \uplus \mathcal{S}_r^2$.

We transform the transition system of $\mathcal{L}^2$ into a Kripke structure by regarding the observation relation O as a proposition and by labelling the states $\langle \gamma_1, \gamma_2\rangle$ of the transition system with O iff $(\gamma_1, \gamma_2) \in O$.

By $\mathcal{K}_{\langle \gamma_1, \gamma_2\rangle}$ we denote the Kripke structure thus constructed, endowed with the single initial state $\langle \gamma_1, \gamma_2\rangle$.

Definition 7.1 *Two configurations γ_1, γ_2 are equivalent, written $\gamma_1 \sim \gamma_2$, if there exists an execution e of the Kripke structure $\mathcal{K}_{\langle \gamma_1, \gamma_2\rangle}$ such that $e \models O \wedge \Box\Diamond O$.*

Example 7.1 *The two 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

$$\gamma_1' \triangleq \langle\langle \texttt{y = 1; y = y+1}\rangle_{\mathsf{k}}\langle \texttt{y} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

are equivalent when O is defined by requiring that `x = y`. *Indeed, in* IMP^2, *starting from $\langle \gamma_1, \gamma_2\rangle$ there is an execution reaching the self-looping state $\langle\langle\langle\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}, \langle\langle\rangle_{\mathsf{k}}\langle \texttt{y} \mapsto 2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\rangle$, which is in O, hence, the execution satisfies $O \wedge \Box\Diamond O$. Note that not all executions of* IMP^2 *starting in $\langle \gamma_1, \gamma_2\rangle$ satisfy $O \wedge \Box\Diamond O$, for example, an execution that applies only rules form $\mathcal{S}_l(\text{IMP}^2)$ followed by self-looping rules violates $O \wedge \Box\Diamond O$.*

Remark 7.2 *The relation O gives us quite a lot of expressiveness for capturing various (deterministic) program equivalences, such as the ones classified in [4]. For example,* partial *equivalence is: two programs are equivalent if, whenever presented with the same input, if they both terminate then they produce the same output. This can be encoded by including cells in the configuration for the input and output, and by including in O the pairs of configurations satisfying: if programs are both empty and inputs are equal then outputs are equal as well. Also,* full *equivalence states that two programs are equivalent if, whenever presented with the same input, they either both terminate and produce the same output, or they both do not terminate. This is captured by adding to the above relation all pairs of configurations from which there exist executions starting from both configurations of the pair, such that the programs in both configurations are forever nonempty. Finally,* reactive *equivalence requires that two programs, when presented with the same sequence of inputs, produce the same sequence of outputs. To encode this equivalence we include in O all configuration pairs satisfying: if the input cells are equal then the output cells are equal as well.*

Remark 7.3 *The chosen definition of equivalence does not work for nondeterministic programs. Indeed, assume a nondeterministic instruction* `|` *such that, for any statements S_1, S_2, the statement S_1* `|` *S_2 rewrites to either S_1 or S_2. Then, the nondeterministic program* $(\texttt{x:=}0)$ `|` $(\texttt{x:=}1)$ *is not equivalent to itself according to our definition (with O being the relation that requires equality of x in both copies of the program). Indeed, one copy of the program can perform $x := 0$*

and self-loop there, while the other one performs $x := 0$ and self-loop there. For the quivalence of nondeterministic programs, the adequate notion of equivalence requires that for all *executions e of one program,* there exists *an execution e' of the other one and an* interleaving *of e, e' satisfying $O \wedge \Box\Diamond O$. This alternation of quantifiers induces additional difficulties for the verification.*

We present in the rest of the section a logic for program equivalence. We present the logic's syntax and a notion of validity for formulas. A *derivative* operation for formulas is also defined.

Definition 7.2 (Syntax) *A* formula *is a pattern of $\mathcal{L}^2$ according to Def. 3.1 applied to $\mathcal{L}^2$, i.e., an expression of the form $\langle \pi_1, \pi_2 \rangle \wedge \phi$ where $\pi_1, \pi_2 \in T_{\Sigma,Cfg}(Var)$ are basic patterns of $\mathcal{L}$ and $C \in T_{\Sigma,Bool}(Var)$.*

Example 7.2 *Assume that the signature Σ for the language* IMP *contains a predicate isModified : Id $\times$ Stmt $\to$ Bool, expressing the fact that the value of the given identifier is modified by the semantics of the given statement. A formula expressing the equivalence of the programs in Example 1.1 is*

$$\left\langle \begin{array}{l} \langle\langle \texttt{for}\, I\ \texttt{from}\, A\ \texttt{to}\ B\ \texttt{do\{}S\,\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle I\ \texttt{=}\ A\,\texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\,\texttt{;}\ I\ \texttt{=}\ I\,\texttt{+1}\,\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I,S) \wedge \neg_{Bool}\ isModified(A,S) \wedge \neg_{Bool}\ isModified(B,S)$$

where M a variable of sort Map. The condition says that the loop counter I is not modified in the body S, and the variables occuring in A, B are not modified by S either. The Boolean function isModified() is defined by structural induction on its arguments in the expected manner.

Recall that the set $[\![f]\!]$, introduced by Definition 3.1 applied to $\mathcal{L}^2$ is the set of configurations $\langle \gamma_1, \gamma_2 \rangle$ of $\mathcal{L}^2$ such that $\langle \gamma_1, \gamma_2 \rangle \in [\![f]\!]$.

Definition 7.3 (Validity) *A formula φ is valid, written $\mathcal{S} \models\!\!\equiv \varphi$, if for all $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$, $\gamma_1 \sim \gamma_2$.*

For the deductive system we shall also be needing the following definition: the *derivative* of a formula f is the set of formulas defined by Definition. 5.2, applied to the symbolic transition of the language $\mathcal{L}^2$. We denote it by $\Delta_{\mathcal{S}^2}(f)$. We let $\Delta_l(f) \triangleq \Delta_{\mathcal{S}^2_l}(f)$ be the left-derivative and $\Delta_r(f) \triangleq \Delta_{\mathcal{S}^2_r}(f)$ be the right-derivative of f. We conclude this section by the following lemma that will be used in proofs regarding our deductive system.

Lemma 7.1 *For all patterns $\varphi \triangleq \langle \pi_1, \pi_2 \rangle \wedge \phi$ of $\mathcal{L}^2$, all instances $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$, and all $h \in \{l, r\}$, there exists $\varphi' \in \Delta_h(\varphi)$ and $\langle \gamma'_1, \gamma'_2 \rangle \in [\![\varphi']\!]$ such that $\langle \gamma_1, \gamma_2 \rangle \Rightarrow_{\mathcal{S}^2} \langle \gamma'_1, \gamma'_2 \rangle$.*

Proof We prove the lemma for $h = l$, the other case being similar. By construction of the language $\mathcal{L}^2$, the pattern φ has the form $\langle \pi_1, \pi_2 \rangle \wedge \phi$, where π_1, π_2 are basic patterns of $\mathcal{L}$, i.e., terms of sort *Cfg* in $\mathcal{L}$, and ϕ is a term of sort *Bool*. Thus, $\pi_1 \wedge \phi$ is a pattern of $\mathcal{L}$. On the other hand, there exists γ'_1 such that $\gamma_1 \Rightarrow_{\mathcal{S}} \gamma'_1$ because the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ has no terminal states (Assumption 2). From $\gamma_1 \Rightarrow_{\mathcal{S}} \gamma'_1$ we obtain $\rho : Var \to \mathcal{T}$ and $\alpha = (l \wedge b \Rightarrow r) \in \mathcal{S}$ such that $\gamma_1 = l\rho$, $\gamma'_1 = r\rho$, and $b\rho = true$. By construction of $\mathcal{L}^2$, there is a rule $\langle l, X \rangle \wedge b \Rightarrow \langle r, X \rangle \in \mathcal{S}^2_l$ for X a variable of sort *Cfg* not ocurring in the rest of the rule, i.e., satisfying $X\rho = X$. By extending ρ into a valuation ρ' such that $X\rho' = \gamma_2$ we obtain the concrete transition $\langle \gamma_1, \gamma_2 \rangle \Rightarrow_{\mathcal{S}^2} \langle \gamma'_1, \gamma_2 \rangle$. Using Lemma 5.1 applied to the transition $\langle \gamma_1, \gamma_2 \rangle \Rightarrow_{\mathcal{S}^2} \langle \gamma'_1, \gamma_2 \rangle$ and the pattern φ we obtain a pattern φ' such that $\langle \gamma'_1, \gamma_2 \rangle \in [\![\varphi']\!]$ and $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}^2_l} \varphi'$, because the rule that is symbolically applied to obtain φ' from φ is $\langle l, X \rangle \wedge b \Rightarrow \langle r, X \rangle \in \mathcal{S}^2_l$. Thus, $\varphi' \in \Delta_l(\varphi)$, which proves the lemma. $\Box$

8 A Circular Proof System

In this section we define a four-rule proof system for proving program equivalence. It is inspired from *circular coinduction* [14], a coinductive proof technique for infinite data structures and coalgebras of expressions [21].

Remember that we have fixed an observation relation O. We assume a set of formulas Ω such that $[\![\Omega]\!] = O$. Let also $\vdash$ be an entailment relation satisfying $\mathcal{S}, F \vdash \varphi$ implies ($\mathcal{S} \models\!\!\equiv \varphi$ or there exists $f \in F$ such that $[\![\varphi]\!] \subseteq [\![f]\!]$). The set Ω and the relation $\vdash$ are parameters of our proof system:

Definition 8.1 (Circular Proof System)

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

$$\textsf{[Reduce]}\ \frac{\mathcal{S}, F \vdash^{\circlearrowleft} G}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}}\ \textit{if}\ \mathcal{S}, F \vdash \varphi$$

$$\textsf{[Circularity]}\ \frac{\mathcal{S}, F \cup \{\varphi\} \vdash^{\circlearrowleft} G \cup \Delta_h(\varphi) \quad h \in \{l, r\}}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}}\ \textit{if}\ [\![\varphi]\!] \subseteq [\![\Omega]\!]$$

$$\textsf{[Derive]}\ \frac{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \Delta_h(\varphi) \quad h \in \{l, r\}}{\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}}\ \textit{if}\ [\![\varphi]\!] \not\subseteq [\![\Omega]\!]\ \textit{and}\ \Delta_h(\varphi) \neq \{\varphi\}$$

An execution *of the proof system is any sequence $\overline{\delta}$ of applications of the above rule. For Γ a set of formulas (also called* goals*), a* proof *of $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ is an execution whose last rule is [Axiom].*

[Axiom] says that when an empty set of goals is reached, the proof is finished. The [Reduce] rule removes from the current set of goals G any goal that can be discharged by the entailment $\vdash$. The last two rules, [Circularity] and [Derive], both say that a goal φ is are replaced by either its left of right derivatives in the set of goals to be proved. However, in [Circularity], the goal φ is added as hypotheses provided that $[\![\varphi]\!] \subseteq [\![\Omega]\!]$, i.e., provided that all its instances are observationally equivalent pairs of configurations. Thus, all the hypotheses f added during executions satisfy $[\![f]\!] \subseteq [\![\Omega]\!] = O$. On the other hand, if $[\![\varphi]\!] \not\subseteq [\![\Omega]\!]$ then the [Derive] rule can be applied, which adds no hypotheses: a goal φ in the current set of goals G is just replaced by its set of left or dight-derivatives, provided that they are not φ itself, i.e., to ensure that [Derive] does not apply uselessly.

The *soundness* of our proof system is the consequence of the following lemmas. By *sequent encountered by* $\overline{\delta}$ we mean any sequent $\mathcal{S}, F \vdash^{\circlearrowleft} G$ which is obtained by applying a prefix of the sequence $\overline{\delta}$ of rules.

Lemma 8.1 *For all sequents $\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}$ encountered by $\overline{\delta}$, for all $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$, there exists a sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ encountered by $\overline{\delta}$ with $[\![\varphi']\!] \subseteq O$, and $\langle \gamma'_1, \gamma'_2 \rangle \in [\![\varphi']\!]$, such that $\langle \gamma_1, \gamma_2 \rangle \Rightarrow^*_{\mathcal{S}^2} \langle \gamma'_1, \gamma'_2 \rangle$.*

Proof By (strong) induction on the length of the proof $\overline{\delta}$ and case analysis. Depending on first the rule of $\overline{\delta}$ that is applied to the sequent $\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}$:

- if the rule is [Reduce] then there are two subcases:
  - if $\mathcal{S} \vdash \varphi$ then $S \models\!\!\equiv \varphi$. Thus, $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!] \subseteq O$, and we let $F' = F$, $G' = G$, $\varphi' = \varphi$, and $\langle \gamma'_1, \gamma'_2 \rangle = \langle \gamma_1, \gamma_2 \rangle$;
  - if $[\![\varphi]\!] \subseteq f$ for some $f \in F$ then $[\![\varphi]\!] \subseteq O$ since all hypotheses f are added (by [Circularity]) such that $[\![f]\!] \subseteq [\![\Omega]\!] = O$, and we can also take $F' = F$, $G' = G$, $\varphi' = \varphi$, and $\langle \gamma'_1, \gamma'_2 \rangle = \langle \gamma_1, \gamma_2 \rangle$;

- if the rule is [Derive] or [Circularity]: using Lemma 7.1, for any $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$, there exists $\varphi'' \in \Delta_h(\varphi)$ and $\langle \gamma_1'', \gamma_2'' \rangle \in [\![\varphi'']\!]$ such that $\langle \gamma_1, \gamma_2 \rangle \Rightarrow_{\mathcal{S}^2} \langle \gamma_1'', \gamma_2'' \rangle$. Moreover, the goal φ'' is the current goal of a future rule application in $\overline{\delta}$ and is the origin of a proof $\overline{\delta}''$ strictly shorter than $\overline{\delta}$. Using the induction hypothesis, there exists $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ and $\langle \gamma_1', \gamma_2' \rangle \in [\![\varphi']\!] \subseteq O$ such that $\langle \gamma_1'', \gamma_2'' \rangle \Rightarrow^*_{\mathcal{S}^2} \langle \gamma_1', \gamma_2'. \rangle$. By transitivity $\langle \gamma_1, \gamma_2 \rangle \in \varphi \Rightarrow^*_{\mathcal{S}^2} \langle \gamma_1', \gamma_2' \rangle$ holds, which proves this case and concludes the proof.

□ The next lemma says that, for each instance of each hypothesis that has *actually been used for discharging a goal* during a proof, there a *strict* successor of it satisfying the current goal of some encountered sequent.

Lemma 8.2 *Let Φ denote the set of all hypotheses used for discharging a subgoal during the proof $\overline{\delta}$ (i.e., using a [Reduce] rule). Then, for all sequents $\mathcal{S}, F \vdash^{\circlearrowleft} G$ encountered by $\overline{\delta}$, for all $f \in F \cap \Phi$, and for all $\langle \gamma_1, \gamma_2 \rangle \in [\![f]\!]$, $P(f, \langle \gamma_1, \gamma_2 \rangle) \triangleq$* there exists $\langle \gamma_1', \gamma_2' \rangle$ and a sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ encountered by $\overline{\delta}$ with $[\![\varphi']\!] \subseteq O$, such that $\langle \gamma_1', \gamma_2' \rangle \in [\![\varphi']\!]$ and $\langle \gamma_1, \gamma_2 \rangle \Rightarrow^+_{\mathcal{S}^2} \langle \gamma_1', \gamma_2' \rangle$ hold.

Proof We show that the lemma's statement holds initially and that it is preserved (as an invariant) by all applications of rules in our deductive system (in particular, for the rules in the proof $\overline{\delta}$). The lemma's statement is trivially true initially, when $F = \emptyset$. For the induction step, we assume that the lemma's statement holds for the current sequent $\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}$ and we show that it holds in the next sequent (if any) in $\overline{\delta}$.

- if the next rule is [Reduce] there are two subcases:
  - $\mathcal{S} \vdash \varphi$. The set $F \cap \Phi$ in the next sequent is the same as in the current one, since this reduction does not use hypotheses in F. With the same instance $\langle \gamma_1', \gamma_2' \rangle$ and sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ given by the inductive hypothesis, we establish that the lemma's statement still holds in the next sequent.
  - $[\![\varphi]\!] \subseteq [\![f_0]\!]$ for some $f_0 \in F$. In this case the set $F \cap \Phi$ in the next sequent is (possibly) larger than in the current one, since this may be the first time the hypothesis f_0 is used to discharge a goal (here, φ). (If $F \cap \Phi$ is the same in the next sequent as in the current one, the inductive hypothesis trivially proves, like in the previous case $\mathcal{S} \vdash \varphi$, that our lemma's statement still holds in the next sequent.)

    Thus, there remains to consider the case where the current rule's application is the first-time use of the hypothesis f to discharge a goal (here, φ), in which case we have to prove $P(f, \langle \gamma_1, \gamma_2 \rangle)$ for all $f \in F \cup \{f_0\}$ and $\langle \gamma_1, \gamma_2 \rangle \in [\![f]\!]$. Now, $P(f, \langle \gamma_1, \gamma_2 \rangle)$ for $f \in F$ and $\langle \gamma_1, \gamma_2 \rangle \in [\![f]\!]$ holds using the inductive hypothesis (this is proved as in the case $\mathcal{S} \vdash \varphi$). There remains to prove $P(f_0, \langle \gamma_1, \gamma_2 \rangle)$ for all $\langle \gamma_1, \gamma_2 \rangle \in [\![f_0]\!]$. For this, we note that f_0 has been added to F at an earlier proof step by [Circularity], and f_0 was replaced in the following sequent's goals by its derivatives $\Delta_h(f_0)$ for some $h \in \{l, r\}$. Using Lemma 7.1, we obtain a goal $f_0'' \in \Delta_h(f_0)$ and $\langle \gamma_1'', \gamma_2'' \rangle \in [\![f_0']\!]$ such that $\langle \gamma_1, \gamma_2 \rangle \Rightarrow_{\mathcal{S}^2} \langle \gamma_1'', \gamma_2'' \rangle$. Using Lemma 8.1 we obtain the instance $\langle \gamma_1', \gamma_2' \rangle$ and the sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ such that $\langle \gamma_1', \gamma_2' \rangle \in [\![\varphi']\!] \subseteq O$ and $\langle \gamma_1'', \gamma_2'' \rangle \Rightarrow^*_{\mathcal{S}^2} \langle \gamma_1', \gamma_2' \rangle$. By transitivity, $\langle \gamma_1, \gamma_2 \rangle \Rightarrow^+_{\mathcal{S}^2} \langle \gamma_1', \gamma_2' \rangle$, which proves that $P(f_0, \langle \gamma_1, \gamma_2 \rangle)$ holds for all $\langle \gamma_1, \gamma_2 \rangle \in [\![f_0]\!]$: the lemma's statement still holds in the next sequent.
- if the next rule is [Circularity] or [Derive]: in this case $F \cap \Phi$ in the next sequent is the same as in the current one, since this rule does not eliminate goals using circular hypotheses (even though, in the case of [Circularity] the current set of hypotheses grows). Like in the case $\mathcal{S} \vdash \varphi$ we establish that the lemma's statement still holds in the next sequent, which concludes this case and completes the proof.

$\square$

The last lemma used for proving our soundness result resembles Lemma 8.1, but it is stronger since it states the existence of *strict* successors in the observation relation. It can be proved thanks to Lemma 8.2.

Lemma 8.3 *For all sequents $\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}$ encountered by $\overline{\delta}$:*

- *either $\mathcal{S} \vdash \varphi$;*
- *or for all $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$, there exists a sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ encountered by $\overline{\delta}$, and $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!] \subseteq O$, such that $\langle\gamma_1, \gamma_2\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle$.*

Proof We proceed by (strong) induction on the length of the proof $\overline{\delta}$ and case analysis. Depending on the first rule of $\overline{\delta}$ that is applied to the sequent $\mathcal{S}, F \vdash^{\circlearrowleft} G \cup \{\varphi\}$:

- if the rule is [Reduce] then there are two subcases:
  - if $\mathcal{S} \vdash \varphi$ then this case is proved;
  - if $[\![\varphi]\!] \subseteq f$ for some $f \in F$: Then, $f \in \Phi$ since f is being used (by the present rule!) to discharge a goal. Thus, $f \in F \cap \Phi$. Using Lemma 8.2 we obtain the sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ and instance $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!] \subseteq O$ such that $\langle\gamma_1, \gamma_2\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle$, which proves this case;
- if the rule is [Derive] or [Circularity]: using Lemma 7.1, for any $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$, there exists $\varphi'' \in \Delta_h(\varphi)$ and $\langle\gamma_1'', \gamma_2''\rangle \in [\![\varphi'']\!]$ such that $\langle\gamma_1, \gamma_2\rangle \Rightarrow_{\mathcal{S}^2} \langle\gamma_1'', \gamma_2''\rangle$. Moreover, the goal φ'' is the current goal of a future rule application in $\overline{\delta}$ and is the origin of a proof $\overline{\delta}''$ strictly shorter than $\overline{\delta}$. Using the induction hypothesis, there exists the sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ and instance $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!] \subseteq O$ such that $\langle\gamma_1'', \gamma_2''\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'.\rangle$. By transitivity $\langle\gamma_1, \gamma_2\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle$ holds, which concludes the proof.

$\square$

Theorem 8.1 (soundness) *Let Γ be a finite set of equivalence formulas. If $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ then $\mathcal{S} \models\!\equiv \Gamma$.*

Proof Pick any $\varphi \in \Gamma$ (if $\Gamma = \emptyset$ the theorem is trivially true). Applying Lemma 8.3 generates two cases:

1. either $\mathcal{S} \vdash \varphi$, which directly implies $\mathcal{S} \models\!\equiv \varphi$;
2. or, for all $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$, there exists a sequent $\mathcal{S}, F' \vdash^{\circlearrowleft} G' \cup \{\varphi'\}$ encountered by $\overline{\delta}$, and $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!] \subseteq O$, such that $\langle\gamma_1, \gamma_2\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle$. We apply Lemma 8.3 to the latter sequent, which generates two cases:
   (a) $\mathcal{S} \vdash \varphi'$, which implies $\mathcal{S} \models\!\equiv \varphi'$. Then from $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!]$, there is an execution satisfying $O \wedge \Box\Diamond O$, and by adding to it the prefix $\langle\gamma_1, \gamma_2\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle$, the resulting execution also satisfies $O \wedge \Box\Diamond O$. Thus, from the (arbitrary) $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$ there is an execution satisfying $O \wedge \Box\Diamond O$, meaning $\mathcal{S} \models\!\equiv \varphi$;
   (b) or for all $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!]$, there exists a sequent $\mathcal{S}, F'' \vdash^{\circlearrowleft} G'' \cup \{\varphi''\}$ encountered by $\overline{\delta}$, and $\langle\gamma_1'', \gamma_2''\rangle \in [\![\varphi'']\!] \subseteq O$, such that $\langle\gamma_1', \gamma_2'\rangle \Rightarrow^{+}_{\mathcal{S}^2} \langle\gamma_1'', \gamma_2''\rangle$. Applying Lemma 8.3 to the latter sequent generates two cases... It is not hard to see that in the first case we will be able to prove $\mathcal{S} \models\!\equiv \varphi$ like in item 2(a) above, and in the second one, another application of Lemma 8.3 will generate yet two more cases...

This repetitive process may never terminate for a goal φ, but it proves $\mathcal{S} \models\!\equiv \varphi$ in one of two possible ways:

- the first one assumes that, after finitely many applications of Lemma 8.3, a subgoal $\varphi^{(n)}$ satisfying $\mathcal{S} \vdash \varphi^{(n)}$ is found. Thanks to Lemma 8.3, from any $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$ there is a finite execution e than leads into some instance $\langle\gamma_1^{(n)}, \gamma_2^{(n)}\rangle \in [\![\varphi^{(n)}]\!]$. And since $\mathcal{S} \models\!\equiv \varphi^{(n)}$, starting from the instance $\langle\gamma_1^{(n)}, \gamma_2^{(n)}\rangle \in [\![\varphi^{(n)}]\!]$, an infinite execution e' satisfying $O \wedge \Box\Diamond O$ exists. The concatenation ee' also satisfies $O \wedge \Box\Diamond O$. Thus, from the arbitrarily chosen $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$ en execution satisfying $O \wedge \Box\Diamond O$ exists, meaning $\mathcal{S} \models\!\equiv \varphi$ holds.
- the second one assumes the contrary: there is no finite number of applications of Lemma 8.3 after which a subgoal $\varphi^{(n)}$ satisfying $\mathcal{S} \vdash \varphi^{(n)}$ is found. In this case, the infinitely many applications of Lemma 8.3 build an infinite execution $\langle\gamma_1, \gamma_2\rangle \Rightarrow^+_{\mathcal{S}^2} \langle\gamma_1', \gamma_2'\rangle \Rightarrow^+_{\mathcal{S}^2} \cdots \Rightarrow^+_{\mathcal{S}^2} \langle\gamma_1^{(n)}, \gamma_2^{(n)}\rangle \Rightarrow^+_{\mathcal{S}^2} \cdots$, starting from any arbitrarily chosen $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!]$, such that O is met infinitely many times, as $\langle\gamma_1, \gamma_2\rangle \in [\![\varphi]\!] \subseteq O$, $\langle\gamma_1', \gamma_2'\rangle \in [\![\varphi']\!] \subseteq O$, $\ldots$, $\langle\gamma_1^{(n)}, \gamma_2^{(n)}\rangle \in [\![\varphi^{(n)}]\!] \subseteq O$, $\ldots$, which implies that our execution satisfies $O \wedge \Box\Diamond O$; thus, $\mathcal{S} \models\!\equiv \varphi$ holds.

In both cases, this process leads to establishing $\mathcal{S} \models\!\equiv \varphi$, and $\varphi \in \Gamma$ was chosen arbitrarily, thus, $\mathcal{S} \models\!\equiv \Gamma$ holds. $\Box$

Remark 8.1 *For soundness it is not essential that the [Circularity] φ actually adds the current goal φ to the current set of circular hypotheses F. What does matter is that, whenever φ is added to F, then $[\![\varphi]\!] \subseteq [\![\Omega]\!]$. We use this observation in our implementation of the proof system to reduce the number of stored hypotheses.*

We now show that the circular proof system, when it terminates, always provides an answer (positive or negative) to the question of whether $\mathcal{S} \models\!\equiv \Gamma$ holds. Thus, in addition to soundness we have a *weak completeness* result. The result is "weak" because it assumes termination of the proof system.

Given a a semantics $\mathcal{S}$ and set of goals Γ, the proof system $\vdash^{\circlearrowleft}$ *terminates successfully* when it returns a proof. The proof system *terminates unsucessfully* when its has a finite, maximal execution that is not a proof - we call such an execution a *disproof.* This happens when the proof system is "stuck": in the current sequence $\mathcal{S}, F \vdash^{\circlearrowleft} G$ no rule of the system can be applied because the side-conditions of the rules are not satisfied. Then, dy definition, the proof system terminates on Γ if it terminates sucessfully or unsucessfully. Weak completeness then says that if a set of goals Γ is valid, all the goals in the set are satisfiable, and the proof system terminates on Γ, then it terminates successfully.

For this we need the following adaptation to the notion of derivative: $\Delta(\varphi) = \{\varphi' \mid \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi' \wedge [\![\varphi']\!] \neq \emptyset\}$, which means that only the satisfiable patterns are kept when computing derivatives. We also need:

Assumption 3 *For all patterns φ of $\mathcal{L}$, if $\Delta_{\mathcal{S}}(\varphi) = \{\varphi\}$ then there is $\pi \Rightarrow \pi \in S$ such that $\Delta_{\mathcal{S}}(\varphi) = \Delta_{\{\pi\Rightarrow\pi\}}(\varphi)$, and for all configurations γ, γ' of $\mathcal{L}$, if $\gamma \Rightarrow_{\mathcal{S}} \gamma$ and $\gamma' \not\Rightarrow_{\mathcal{S}} \gamma'$ then $\langle\gamma, \gamma''\rangle \notin O$.*

Both assumptions regard the language $\mathcal{L}$ of interest. The first says that, whenever a the derivative of pattern is the pattern itself, then the only rule that contributes to this derivative is of the form rules $\pi \Rightarrow \pi$. Remember (Assumption 2) that such rules were included in the semantics $\mathcal{S}$ for technical reasons in order to transform terminal configurations into self-looping ones (ultimately, because we deal with LTL over infinite sequences). Our first asssumption then says that, except for the rules, $\pi \Rightarrow \pi$ were added to the semantics, all the other rules "change" at least "something" in a pattern; ie., rules that do not change anything in the semantics of a language are useless. Regarding the second of the above assumptions, it says that self-looing configurations and non self-looping ones cannot be observationally equivalent. As observed before, the self-looping configurations are (formerly) terminal configurations that were transformed into self-looping ones by including the rules of the form $\pi \Rightarrow \pi$ in the semantics $\mathcal{S}$. Thus, our second assumption actually says that configurations where the code to be executed is finished, and configurations where there is still code to execute, cannot be observationally equivalent, which is also a reasonable constraint on equivalence.

Theorem 8.2 (weak completeness) *If $\mathcal{S} \models \Gamma$ and for all $\varphi \in \Gamma$ it holds that $[\![\varphi]\!] \neq \emptyset$, and the $\vdash^{\circlearrowleft}$ proof system terminates on Γ then $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$.*

Proof By contradiction: assume the hypotheses hold but not the conclusion, i.e., $\mathcal{S} \not\vdash^{\circlearrowleft} \Gamma$. Thus, the proof system terminates with a disproof $\overline{\delta}$, i.e., a sequence of rule applications that is not a proof and after which no rule can be applied. Let $\mathcal{S}, F \vdash^{\circlearrowleft} G$ be the sequent resulting after $\overline{\delta}$. Thus, $G \neq \emptyset$, and for all $\varphi \in G$, $[\![\varphi]\!] \not\subseteq O$ (otherwise, [Circularity] would be applicable), and $\Delta_h(\varphi) = \{\varphi\}$ for $h \in \{l, r\}$ (otherwise, [Derive] would be applicable). We choose any $\varphi \in G$. Since both [Circularity] and [Derive] rules compute derivatives, there exists $\varphi_0 \in \Gamma$ and a symbolic execution $\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}^2} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}^2} \varphi_n = \varphi$. The symbolic execution is feasible, since we have assumed that only satisfiable patterns are kept in the derivatives. Moreover, $[\![\varphi_n]\!] \not\subseteq O$, thus, we can choose $\langle \gamma, \gamma' \rangle \in [\![\varphi]\!] \setminus O$. Hence, we can apply Corollary 5.2 and find a concrete execution $\langle \gamma_0, \gamma_0' \rangle \Rightarrow_{\mathcal{S}^2} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}^2} \langle \gamma_n, \gamma_n' \rangle = \langle \gamma, \gamma' \rangle$ such that for all $i = 0, n-1$, $\langle \gamma_i, \gamma_i' \rangle \in [\![\varphi_i]\!]$, and $\langle \gamma_n, \gamma_n' \rangle \in [\![\varphi_n]\!] \setminus O$.

Next, due to the definition of the language $\mathcal{L}^2$, by *projecting* the above concrete execution of $\mathcal{L}^2$ on its left and right components we obtain the two executions $e \triangleq \gamma_0 \Rightarrow^*_{\mathcal{S}} \gamma$ and $e' \triangleq \gamma_0' \Rightarrow^*_{\mathcal{S}} \gamma'$ of $\mathcal{L}$. Let $\varphi = \langle \pi, \pi' \rangle \wedge \phi$, then, $\gamma \in [\![\pi \wedge \phi]\!]$ and $\gamma' \in [\![\pi' \wedge \phi]\!]$. From $\Delta_l(\varphi) = \Delta_r(\varphi) = \{\varphi\}$ we obtain $\Delta_{\mathcal{S}}(\pi \wedge \phi) = \{\pi \wedge \phi\}$ and $\Delta_{\mathcal{S}}(\pi' \wedge \phi) = \{\pi' \wedge \phi\}$, thus, Using Assumption 3, both these derivarives were obtained by applying rules of the form $\pi \Rightarrow \pi \in \mathcal{S}$. Thus, there are transitions $\gamma \Rightarrow_{\mathcal{S}} \gamma$ and $\gamma' \Rightarrow_{\mathcal{S}} \gamma'$ in $\mathcal{L}$, and the finite executions e, e' can be extended into infinite ones $\overline{e} \triangleq \gamma_0 \Rightarrow^*_{\mathcal{S}} \gamma \Rightarrow_{\mathcal{S}} \gamma \cdots \Rightarrow_{\mathcal{S}} \gamma \cdots$ and $\overline{e}' \triangleq \gamma_0' \Rightarrow^*_{\mathcal{S}} \gamma' \Rightarrow_{\mathcal{S}} \gamma' \cdots \Rightarrow_{\mathcal{S}} \gamma' \cdots$, and since $\mathcal{L}$ is deterministic, $\overline{e}$, resp. $\overline{e}'$ is the only infinite execution of $\mathcal{L}$ starting in γ_0, resp. in γ_0'.

Moreover, *the infinite executions in $\mathcal{L}^2$ starting in $\langle \gamma_0, \gamma_0' \rangle$ coincide with sequences obtained by interleaving transitions of $\overline{e}$ and $\overline{e}'$*. It is thus enough consider any such interleaving, denoted hereafter by $\overline{e} \amalg \overline{e}'$, and show that it satisfies $\Diamond\Box\neg O$.

There are two cases:

- in $\overline{e} \amalg \overline{e}'$, both $\overline{e}$ and $\overline{e}'$ have reached γ, resp. γ'. Thus, $\overline{e} \amalg \overline{e}'$ has an infinite suffix that only repeats the instance $\langle \gamma, \gamma' \rangle \in [\![\varphi]\!] \setminus O$, which is does not satisfy O; thus, $\overline{e} \amalg \overline{e}' \models \Diamond\Box\neg O$.

- in $\overline{e} \amalg \overline{e}'$, only one of the components, say, e, has reached γ. Thus, $\overline{e} \amalg \overline{e}'$ has an infinite suffix that only repeats an instance of the form $\langle \gamma, \gamma'' \rangle$, for some configuration γ'' that does not have a transition $\gamma'' \Rightarrow_{\mathcal{S}} \gamma''$ (that would contradict the unicity of the infinite execution $\overline{e}'$ starting in γ_0'). By Assumption 3, we have $\langle \gamma, \gamma'' \rangle \notin O$ and since $\overline{e} \amalg \overline{e}'$ ends up by only repeating $\langle \gamma, \gamma'' \rangle$, we have $\overline{e} \amalg \overline{e}' \models \Diamond\Box\neg O$ again.

Recapitulating, we have obtained a goal $\varphi_0 \in \Gamma$ and an instance $\langle \gamma_0, \gamma_0' \rangle \in [\![\varphi_0]\!]$, and two infinite executions $\overline{e}$ and $\overline{e}'$ of $\mathcal{L}$, starting in γ_0, and γ_0', respectively, such that any infinite execution of $\mathcal{L}^2$ starting in $\langle \gamma_0, \gamma_0' \rangle$ is of the form $\overline{e} \amalg \overline{e}'$ for some interleaving $\amalg$ of e, e'; and any such infinite execution $\overline{e} \amalg \overline{e}'$ satisfies $\Diamond\Box\neg O$. According to Definition 7.1 this means $\gamma_0 \not\sim \gamma_0'$, and according to Definition 7.3, this means $\mathcal{S} \not\models \varphi_0$. Hence, $\mathcal{S} \not\models \Gamma$, which is in contradiction to the hypothesis

$\mathcal{S} \models \Gamma$ of our theorem. The contradiction was obtained by assuming $\mathcal{S} \not\vdash^{\circlearrowleft} \Gamma$, hence, $\mathcal{S} \vdash^{\circlearrowleft} \Gamma$ holds, which concludes the proof. $\square$

Together, the soundness and weak completeness say results 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 as we have stated it. 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 does not change the goals and no rule can be applied any more) and also for goals in which the programs does not terminate, but behave in a certain "regular" way, as shown in the examples below.

Example 8.1 *We start by illustrating the use of the deductive system on the equivalence of* STREAM *programs since it does not require unification, hence it is a bit easier. The equivalence we want to prove is that from Example 1.2:* `blink` *is equivalent to* `zip(zero, one)`*. This is written as the equivalence formula*

$$\left\langle \begin{array}{l} \langle\langle \texttt{blink}\rangle_{\mathsf{k}}\langle spec_1\rangle_{\mathsf{specs}}\langle Y_1\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}, \\ \langle\langle \texttt{zip(zero, one)}\rangle_{\mathsf{k}}\langle spec_2\rangle_{\mathsf{specs}}\langle Y_2\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\rangle \end{array} \right\rangle$$
$$\wedge\, Y_1 = Y_2 \tag{2}$$

where $spec_1$ *is* `blink` $\mapsto \lambda()$ `.0:1:blink` *and* $spec_2$ *is the map*

$$\begin{array}{l} \texttt{zero} \mapsto \lambda()\ \texttt{.0:zero} \\ \texttt{one} \mapsto \lambda()\ \texttt{.1:one} \\ \texttt{zip} \mapsto \lambda(xs, ys)\ \texttt{.zip(}\ ys\ \texttt{,}\ \texttt{hd(}\ xs\ \texttt{))} \end{array}$$

Note that the contents of the cells specs *is not changed during the execution of the program. The observation relation is given by*

$$\Omega = \{\langle\langle\langle C_1\rangle_{\mathsf{k}}\langle spec_1\rangle_{\mathsf{specs}}\langle Y_1\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}, \langle\langle C_2\rangle_{\mathsf{k}}\langle spec_2\rangle_{\mathsf{specs}}\langle Y_2\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\rangle\rangle\} \wedge Y_1 = Y_2$$

where C_1 *and* C_2 *are two arbitrary* STREAM *programs. In words, two configurations are observational equivalent iff the corresponding output cells* out *have equal contents.*

The equivalence formula (2) is the unique goal in G *we start with. We first apply* [Circularity] *for the program* `blink` *(i.e., in the proof system, the derivative* $\Delta_l()$ *is applied), which loads the definition of* `blink` *in the* k *cell, and adds (2) to the set of circular hypotheses* F*. We then apply* [Derive] *twice, which writes in the corresponding output cell the two head elements of the stream, and produces the following goal:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{blink}\rangle_{\mathsf{k}}\langle spec_1\rangle_{\mathsf{specs}}\langle \texttt{0 : 1 :}\ Y_1\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{zip(zero, one)}\rangle_{\mathsf{k}}\langle spec_2\rangle_{\mathsf{specs}}\langle Y_2\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\rangle \end{array} \right\rangle$$
$$\wedge\ Y_1 = Y_2 \tag{3}$$

Not that the contents of the output cell in the first configuration has changed. Next, by applying Derive *several times with the heating/cooling rules that compute the arguments of* `zip(zero, one)`*, we get*

$$\left\langle \begin{array}{l} \langle\langle \texttt{blink}\rangle_{\mathsf{k}}\langle spec_1\rangle_{\mathsf{specs}}\langle \texttt{0 : 1 :}\ Y_1\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{zip(0 : zero, 1 : one)}\rangle_{\mathsf{k}}\langle spec_2\rangle_{\mathsf{specs}}\langle Y_2\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\rangle \end{array} \right\rangle$$
$$\wedge\ Y_1 = Y_2 \tag{4}$$

Several other applications of Derive *proceed with loading the definition of* `zip` *in the* k *cell, applying heating/cooling rules for* `hd(zero)`*, adding content to the output cell, and computing new* *arguments of* **zip***:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{blink} \rangle_{\mathsf{k}} \langle spec_1 \rangle_{\mathsf{specs}} \langle \texttt{0 : 1 :} \ Y_1 \rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{zip(1 : one, 0 : zero)} \rangle_{\mathsf{k}} \langle spec_2 \rangle_{\mathsf{specs}} \langle \texttt{0 :} \ Y_2 \rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}} \rangle \end{array} \right\rangle$$
$$\wedge \ Y_1 = Y_2 \tag{5}$$

The above process is repeated with the new instance of **zip***, which produces the following goal:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{blink} \rangle_{\mathsf{k}} \langle spec_1 \rangle_{\mathsf{specs}} \langle \texttt{0 : 1 :} \ Y_1 \rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{zip(0 : zero, 1 : one)} \rangle_{\mathsf{k}} \langle spec_2 \rangle_{\mathsf{specs}} \langle \texttt{0 : 1 :} \ Y_2 \rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}} \rangle \end{array} \right\rangle$$
$$\wedge \ Y_1 = Y_2 \tag{6}$$

To conclude the proof, we note that (6) is an instance of (2) by applying the substitution $\{Y_1 \mapsto \texttt{0 : 1} \ Y_1, Y_2 \mapsto \texttt{0 : 1} \ Y_2\}$*. Hence,* $[\![(6)]\!] \subseteq [\![(2)]\!]$*, and the* Reduce *discharges the (unique) current goal (6), and* Axiom *concludes the proof.*

Example 8.2 *We show the application of our proof system for proving the equivalence of* `for` *and* `while` *programs formalised as the validity of the following formula, with* $A, B : Int$*,* $S : Stmt$*,* $I : Id$ *and* $M : Map$ *Considering* A, B *to be integers instead of expressions is not a restriction, since, if* A *and* B *were artihmetical expressions, the strictness attributes for the* `for`*, assigment, and* `<=` *would be applied first and would transform* A, B *into integers anyway. This allows us to simplify the original equivalence formula , given in Example 7.2, into the following one, based on the fact that* $isModified(A, S) = isModified(B, S) = false$*:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}S\texttt{\}} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \\ \langle\langle I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \texttt{\}} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge \ \neg_{Bool} \ isModified(I, S) \tag{7}$$

The observation relation is given by the set $\Omega = \{\langle\langle\langle C_1 \rangle_{\mathsf{k}} \langle M' \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}, \langle\langle C_2 \rangle_{\mathsf{k}} \langle M'' \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \rangle \wedge M' =_{Map} M''\}$*. The relation says that two configurations are observationally equivalent iff they have equal environments.*

In order to prove the goal (7) with our proof system we start with a set of goals G *consisting of (7) and*

$$\left\langle \begin{array}{l} \langle\langle C \curvearrowright (\texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}S\texttt{\})} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \\ \langle\langle C \curvearrowright (I\ \texttt{=}\ I\ \texttt{+}1\texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \texttt{\})} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge \ \neg_{Bool}\ isModified(I, C) \wedge_{Bool} lookup(M, I) = A \tag{8}$$

where C *is a variable of sort Code. Remember that Code is a sort that includes all statements and arithmetical and Boolean expressions, that* $\cdot$ *denotes the empty code, and that code sequencing is denoted by* $\curvearrowright$*.*

In the sequel we show the application of the rules of our proof system to the chosen set of goals G*. The first rule applied to (7) is* [Circularity]*, by which (7) is added to the hypotheses* H *and is replaced by a goal obtained by applying the semantical rule for the* `for` *statement, which gives:*

$$\left\langle \begin{array}{l} \langle\langle I\ \texttt{=}\ A\ \texttt{;if}\ I\ \texttt{<=}\ B\ \texttt{then}\ S\ \texttt{;for}\ \ I\ \ \texttt{from}\ \ A +_{Int} 1\ \ \texttt{to}\ \ B\ \texttt{do}\ \{S\}\ \texttt{else skip}\} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \\ \langle\langle I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \texttt{\}} \rangle_{\mathsf{k}}, \langle M \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge \ \neg_{Bool} \ isModified(I, S)$$

We now apply the sequence of rules [Circularity], [Derive], [Circularity], without adding new hypotheses[5]*, which replaces the above goal with the following one, obtained by applying the semantics of assignment to both sides of the formula and then the semantical rule for the* `while` *statement:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{if}\ I \ \texttt{<=} B\ \texttt{then} S\ \texttt{;for}\ I\ \texttt{from}\ A +_{Int} 1\ \texttt{to}\ B\ \texttt{do}\ \{S\}\ \texttt{else}\ \texttt{skip}\}\rangle_{\mathsf{k}}, \langle update(M,I,A\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{if}\ I \ \texttt{<=} B\ \texttt{then}\ S\ \texttt{;}\ I\ \texttt{=} I\ \texttt{+}1\ \texttt{;while}\ I \ \texttt{<=} B\ \texttt{do\{} S\ \texttt{;}\ I\ \texttt{=} I\ \texttt{+}1\ \texttt{\}else skip}\rangle_{\mathsf{k}}, \langle update(M,I,A\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge \neg_{Bool}\ isModified(I,S)$$

Next[6]*, the heating rules for the* `if` *statement and the* `_ <= _` *operation, followed by the cooling rules, and finally the rules that conclude the evaluation of the* `if` *statement result in the two following subgoals:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{skip}\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{skip}\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I,S) \wedge_{Bool} \neg_{Bool} A \leq_{Int} B$$

$$\left\langle \begin{array}{l} \langle\langle S\ \texttt{;for}\ I\ \ \texttt{from}\ \ A +_{Int} 1\ \ \texttt{to}\ \ B\ \texttt{do}\ \{S\}\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \ \texttt{; while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \texttt{\}}\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I,S) \wedge_{Bool}\ A \leq_{Int} B$$

The first subgoal is trivially valid and is eliminated by the [Reduce] rule using the base entailment $\vdash$*.*
By applying the semantical rule for statement sequencing, which rewrites `;` *to* $\curvearrowright$*, for the second one, we get a new goal*

$$\left\langle \begin{array}{l} \langle\langle S \curvearrowright (\texttt{for}\ \ I\ \ \texttt{from}\ \ A +_{Int} 1\ \ \texttt{to}\ \ B\ \texttt{do}\ \{S\})\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle S \curvearrowright (I\ \texttt{=}\ I\ \texttt{+}1\ \ \texttt{; while}\ I\ \texttt{<=}\ B\ \texttt{do}\ \{S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\})\rangle_{\mathsf{k}}, \langle update(M,I,A)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I,S) \wedge_{Bool}\ A \leq_{Int} B \tag{9}$$

which is eliminated by the the [Reduce] rule since it is an instance of the goal (8) (by using the substitution $C \leftarrow S, M \leftarrow update(M,I,A)$*, and by using the equality* $lookup(I, update(M,I,A)) = A$*).*

To conclude the proof we also need to eliminate the goal (8). This elimination amounts to unifying the code C *with all possible left-hand sides of rules in the semantics of* IMP*. We only give a subset of all the cases, since considering all cases may be overlong for the reader's patience (but not so for a computer). We first consider the case where* S *is unified with statements:*

- $C \leftarrow$ `skip`*: by applying the semantical rules for* `skip` *(which rewrites it to* $\cdot$*), then the rule that consumes the empty code* $\cdot$*, and finally the rule sequence that evaluates* `I + 1` *in the goal's right-hand side, the goal (8) becomes the following one, which is implied by the initial goal (7) and is eliminated by [Reduce]:*

$$\left\langle \begin{array}{l} \langle\langle \texttt{for}\ I\ \texttt{from}\ A\ \texttt{to}\ B\ \texttt{do\{}S\texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle I\ \texttt{=}\ A\ \texttt{;while}\ I\ \texttt{<=}\ B\ \texttt{do\{}S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+}1\ \texttt{\}}\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I,S) \wedge_{Bool} lookup(M,I) = A$$

[5]which is sound thanks to Remark 8.1. In the sequel, whenever [Circularity] is applied, by default it does not add new hypotheses.

[6]In the sequel we mention only the semantical rules used in the sequence of rules [Circularity] and [Derive] that is applied to obtain the next goal.

- $C \leftarrow \{S_1 \texttt{ ; } S_2\}$ *for some statements* S_1, S_2*: the rule rewriting* ; *to* $\curvearrowright$ *produces an instance of the goal (8) itself, with the substitution* $C \leftarrow S_1 \curvearrowright S_2$*, which is then eliminated by [Reduce].*
- $C \leftarrow \{S'\}$ *for some statement* S'*: the rule for* $\{\_\}$ *elimination produces an instance of the goal (8) itself, with the substitution* $C \leftarrow S'$*, which is then eliminated by [Reduce].*
- $C \leftarrow \texttt{if}\ B'\ \texttt{then}\ S_1\ \texttt{else}\ S_2$*, for some Boolean expression* B' *and statements* S_1, S_2*: there are two subcases, depending on whether* B' *has the sort Bool, or does not have the sort Bool but has sort BExp:*
  - *if* B' *has the sort Bool then one can directly apply the rules for* `if` *and obtain two subgoals: one is*

$$\left\langle \begin{array}{l} \langle\langle S_1 \curvearrowright (\texttt{for}\ I\ \texttt{from}\ A +_{Int} 1\ \texttt{to}\ B\ \texttt{do}\ \{S\})\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle S_1 \curvearrowright (I\ \texttt{=}\ I\ \texttt{+1}\ \texttt{;}\ \texttt{while}\ I\ \texttt{<=}\ B\ \texttt{do}\ \{S\ \texttt{;}\ I\ \texttt{=}\ I\ \texttt{+1}\})\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I, S_1) \wedge_{Bool} \neg_{Bool}\ isModified(I, S_2) \wedge_{Bool}\ B' =_{Bool} true$$

    (where we used $isModified(I, \texttt{if}\ B'\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = isModified(I, S_1) \vee_{Bool} isModified(I, S_2)$*). This is an instance of the goal (8) and is eliminated by [Reduce]*[7]*. The other subgoal is similar, but with* S_2 *instead of* S_1 *and* $B' =_{Bool} false$ *in the condition, which is also an instance of the goal (8).*
  - *if* B' *does not have the sort Bool then it has the sort BExp. Then, the only rule that our goal can be unified with is the heating rule for* `if`*, which generates the following goal:*

$$\left\langle \begin{array}{l} \langle\langle (B' \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \curvearrowright (\texttt{for}\ I\ \texttt{from}\ A +_{Int} 1\ \texttt{to}\ B\ \texttt{do}\ \{S\})\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\ \langle\langle (B' \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \curvearrowright (I\ \texttt{=}I\ \texttt{+1}\ \texttt{;while}\ I\ \texttt{<=}B\ \texttt{do}\ \{S\texttt{;}\ I\ \texttt{=}I\ \texttt{+1}\})\rangle_{\mathsf{k}}, \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \end{array} \right\rangle$$
$$\wedge\ \neg_{Bool}\ isModified(I, S_1) \wedge_{Bool} \neg_{Bool}\ isModified(I, S_2)$$

    which is an instance of (8) with $S \leftarrow (B' \curvearrowright \texttt{if}\ \square\ \texttt{then}\ S_1\ \texttt{else}\ S_2)$.
- $C \leftarrow \texttt{while}\ B'\ \texttt{do}\ S'$*. The rule for* `while` *transforms (8) into an instance of itself under the substitution* $C \leftarrow (\texttt{if}\ B'\ \texttt{then}\ S\ \texttt{;}\ I\ \texttt{+}1\ \texttt{;}\ \texttt{while}\ B'\ \texttt{do}\ S'\ \texttt{else skip})$.
- $C \leftarrow \texttt{for}\ I'\ \texttt{from}\ A'\ \texttt{to}\ B'\ \texttt{do}\ S'$*. The rule for* `for` *transforms (8) into an instance of itself under the substitution* $C \leftarrow (I'\ \texttt{=}\ A'\ \texttt{;}\ \texttt{if}\ B'\ \texttt{then}\ S\ \texttt{;}\ I\ \texttt{+}1\ \texttt{;}\ \texttt{for}\ I'\ \texttt{from}\ A'\ \texttt{to}\ B'\ \texttt{do}\ S'\ \texttt{else skip})$.
- $C \leftarrow X$ *for some identifier* X*, which amounts to unification with the rule for program-variable lookup. That rule transforms our goal into an instance of itself with* $C \leftarrow lookup(M, X, I)$.
- $C \leftarrow X' = A'$ *for some identifier* X' *and arithmetical expression* A'*. Similar to the case of* `if`*, there are subcases depending on whether* $'$ *has sort Int, or does not have sort Int but has sort AExp.*
  - *in the first case the rule for variable assignment transforms (8) into an instance of itself with* $C \leftarrow \cdot$ *and* $M \mapsto update(M, X, I)$*;*
  - *in the second case, the heating rule for variable assignment transforms 8) into an instance of itself with* $C \leftarrow (A' \curvearrowright I = \square)$.

[7] in the sequel, whenever (8) is transformed into an instance of itself, we omit the sentence "and is eliminated by [Reduce]".

There remain to consider the cases where C is code but is not a statement. The goal (8) can be unified with left-hand sides of semantical rules:

- $C \leftarrow C_1 \curvearrowright C_2$*: then unification may be performed with both heating and cooling rules.*
  - *We first illustrate the situation with the cooling rule for the* `if` *statement, which was explicitly given in Section 2.1; the case for all the other cooling rules is completely similar. In the considered case,* $C_1 \leftarrow B$ *and* $C_2 \leftarrow$ `if` $\square$ `then` S_1 `else` $S_2 \curvearrowright C'$ *for some code* C'*, and the cooling rule transforms (8) into an instance of itself with* $C \leftarrow$ `if` B `then` S_1 `else` $S_2 \curvearrowright C'$.
  - *Regarding unification with heating rules, this may only happen when the left-hand side of the rule is of the form* $\langle\langle C_1' \curvearrowright C_2'\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$*, and the right-hand side has the form* $\langle\langle(C_1'' \curvearrowright C_2'') \curvearrowright C_2'\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$*; the application of this rule transforms (8) into an instance of itself with* $C \leftarrow C_1'' \curvearrowright C_2''$.
- C *is an arithmetical expression or a Boolean expression. Then, again, unification may be performed with both heating and cooling rules, in a completely similar many to what has been shown above.*

Thus, in all possible cases, the goal goal (8) is transformed into an instance of itself and is eliminated from the set of goals. Since the other goal (7) has been eliminated earlier, the proof system terminates sucessfully.

9 A Prototype Implementation

$\mathbb{K}$ [16] 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 [22] specifications. Programs of languages defined in $\mathbb{K}$ can thus be executed and analysed using Maude as the underlying rewriting engine. $\mathbb{K}$ also offers some support for symbolic computations, including a connection to the Z3 SMT solver [23]. We have used these components in a prototype tool implementing our deductive system for program equivalence. Here we describe how the proposed proof system is implemented for the IMP and STREAM languages. This description is generic enough and can be seen as a methodology applicable to any language defined in $\mathbb{K}$.

There are (at least) two approaches to implementing the proof system:

1. as an external procedure, which uses the $\mathbb{K}$ tool for computing derivatives of equivalence formulas only. The external procedure is then responsible all the other operations, including the searching for proofs;
2. directly in $\mathbb{K}$, by performing all the operations in the proof system using the available $\mathbb{K}$ tools (for example, the underlying Maude search engine is used in searching for proofs). This requires extending the definition of the language of interest definitions with additional data structures, with semantical rules for storing circular hypotheses, and with rules for the entailment between these hypotheses and goals.

Since our approach is parametric in the language definition, observational relation, and basic entailment, in both cases we need a procedure that builds the definition of $\mathcal{L}^2$ for a given $\mathcal{L}$, and procedures for the basic entailment ($\mathcal{S} \vdash \varphi$) and subsumption ($[\![\varphi]\!] \subseteq [\![f]\!]$). The basic entailment relation $\mathcal{S}^2 \vdash \varphi$ can be specified by means of a (finite) set of equivalence formulas $\mathcal{E}$ (in the same way that Ω specifies the observation relation O), and taking $\mathcal{S}^2 \vdash \varphi$ iff there is $e \in \mathcal{E}$ such that $[\![\varphi]\!] \subseteq [\![e]\!]$. The subsumption relation can be checked using Proposition 3.2 or Proposition 3.3.

For IMP the set Ω will typically consist of formulas that say that a given set of program variables have the same values in both configurations, and $\mathcal{E}$ further requires that the two contents of the k cells are the same. For STREAM, Ω says that the two out cells have the same contents.

By Proposition 3.1, the formulas $f \in F$ and $e \in \mathcal{E}$ can always be stored in the form $\pi' \wedge \bigwedge_\sigma \wedge \phi$, which facilitates the checking of subsumptions based on Proposition 3.2 or Proposition 3.3. The validity of the implication from Proposition 3.2 is checked by calling the Z3 SMT solver. The substitution σ (ocurring in formulas of the form $\pi' \wedge \bigwedge_\sigma \wedge \phi$) is computed by inspecting the contents of the two configurations π and π'.

We chose to implement the proof system for the two languages directly in $\mathbb{K}$ since this is the most straightforward approach and allows us to benefit from tools in the $\mathbb{K}$ framework. However, we had to make some compromises. Since the current Maude backend of $\mathbb{K}$ is a rewriting engine based on matching, we had to axiomatise symbolic statements instead of using unification for them. The main axiom says how a symbolic-statement variable S affects the environment M under a current condion ϕ:

$$\langle\langle S \cdots\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \phi \Rightarrow \langle\langle \cdot \cdots\rangle_{\mathsf{k}}\langle \mathit{followup}(S, M, \phi)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \phi$$

The function *followup* is axiomatised as well; its axiom says that S has no effect on a variables X that it does not modify: $\mathit{followup}(S, (X \mapsto V\ M), \phi) = X \mapsto V\ \mathit{followup}(S, M, \phi)$ when ϕ implies $\neg \mathit{isModified}(X, S)$.

An equivalence formula $\varphi \triangleq \langle \pi_1, \pi_2\rangle \wedge \phi$ for IMP is written in $\mathbb{K}$ as an IMP configuration

$$\langle\langle p_1\rangle_{\mathsf{k}}\langle M_1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_1}}\langle\langle p_2\rangle_{\mathsf{k}}\langle M_2\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_2}}\langle \phi\rangle_{\mathsf{cond}}$$

where the pattern π_i is given by the contents of the $\mathsf{cfg_i}$ cell and the condition ϕ is stored into a new cell called cond. The circularities F are stored into a new cell hypos. For each circularity $f \in F$, the subsumption relation $[\![\varphi]\!] \subseteq [\![f]\!]$ is checked by means of two substitutions. The first one is a substitution σ from the contents of the cell k in f to the coresponding one in φ. Let

$$f \triangleq \langle\langle p_1'\rangle_{\mathsf{k}}\langle M_1'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_1}}\langle\langle p_2'\rangle_{\mathsf{k}}\langle M_2'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_2}}\langle \phi'\rangle_{\mathsf{cond}}$$

such an hypothesis in F and let φ be the current equivalence formula represented as above. For instance, if φ is given by (9) and f by (8), then σ is $A \leftarrow A +_{Int} 1$. The expressions from the codomain of σ are evaluated in the current configuration; in this way, e.g., the program variables are replaced by their current values. Since the cell env includes only fresh variables, we have $f\sigma$ equal to

$$\langle\langle p_1\rangle_{\mathsf{k}}\langle M_1'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_1}}\langle\langle p_2\rangle_{\mathsf{k}}\langle M_2'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg_2}}\langle \phi'\sigma\rangle_{\mathsf{cond}}$$

The second substitution σ' is between the corresponding env cells such that $M_i'\sigma' = M_i$ for $i = 1, 2$. Note that $f\sigma\sigma' = \varphi$ does not hold in general, because usually $\phi'\sigma\sigma' = \phi$ does not. But if $\phi \wedge \bigwedge_{\sigma'}$ implies $\phi'\sigma$ holds, which is checked by calling the SMT solver, then we obtain $[\![\varphi]\!] \subseteq [\![f\sigma]\!]$ by Proposition 3.2 . Since $[\![f\sigma]\!] \subseteq [\![f]\!]$ by Proposition 3.3, it follows that $[\![\varphi]\!] \subseteq [\![f]\!]$.

This method for checking the subsumption relation is not specific to IMP. For any other language definition the substitution σ is defined by structural induction on the language syntax and the substitution σ' is computed by considering the rest of configurations. For instance, for the case of STREAM, only the substitution σ is required because the rest of configurations remain constant during the execution.

The efficiency of the implementation depends on how the [Circularity] and [Derive] rules are applied. In Remark 8.1 we noted that, for soundness, it is not necessary to always add the current goal φ to the hypotheses F when applying [Circularity]. Ideally, only those formulas actually subsequently used by [Reduce] rules should be added. Since there is no way of knowing in advance which circular hypotheses will be used in the future, we apply a heuristic when adding circular hypotheses. This is achieved by using labelled statements: each time two statements with the same label are at the top of the `k` cells, a set of rules decides which one of the following three cases holds for the current configuration and takes the corresponding action: whether it belongs to the observation relation, or it is a consequence of the circular hypotheses, or it must be added to the circular hypotheses.

The contents of the cell hypos at the end of a successful proof includes in fact new equivalences that the tool discovered during proving process. For instance, if the goal is $\mathtt{morse} \approx \mathtt{f(morse)}$, where

$$\mathtt{morse} \approx \mathtt{0{:}1{:}zip(tl(morse),\ not(tl(morse)));} \qquad \mathtt{not(}\ xs\ \mathtt{)} \approx \mathtt{neg(hd(}\ xs\ \mathtt{)){:}not(}\ xs\ \mathtt{);}$$

$$\mathtt{f(}\ xs\ \mathtt{)} \approx \mathtt{hd(}\ xs\ \mathtt{){:}neg(hd(}\ xs\ \mathtt{)){:}f(tl(}\ xs\ \mathtt{));} \qquad \mathtt{neg(}\ x\ \mathtt{)\ :=\ 1} \lhd x =_{Int} 0 \rhd \mathtt{0}$$

then the following new equivalence is found:

$$\mathtt{1{:}f(1{:}zip(tl(morse),\ not(tl(morse))))} \approx \mathtt{1{:}zip(tl(morse),not(tl(morse)))}.$$

We also note that the STREAM example shows that the proof system introduced in this paper includes the one defined in [14] whenever behavioural equational specifications can be encoded as programming language definitions. However, the definition for the equivalence we introduced here is more general: the equivalence considered in [14] can be defined using the LTL formula pattern $\Box O$ while the one defined here uses $\Box\Diamond O$.

10 Conclusion and Future Work

We have presented a definition for program equivalence, a logic that encodes this definition in its formulas, and a proof system for the logic, which is proved sound and weakly complete. A prototype implementation for the proof system in the $\mathbb{K}$ framework was also presented and illustrated on example of equivalent programs in languages from two different paradigms.

The proposed approach is generic: it does not depend on $\mathbb{K}$ and the language being defined in $\mathbb{K}$, but requires a formal semantics of the language of interest as a term-rewriting system. The chosen equivalence relation is also parametric in a certain observation relation and requires that starting from configurations in the observartion relation, configurations in the observarion relation will be encountered again. We show the approach is applicable for concrete and symbolic programs and for terminating and non-terminating ones.

Future Work We are currently applying 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, dynamical function definitions). 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. We are also planning to combine our program-equivalence verification with matching logic [19], a language-independent logic for programs written in languages with a rewrite-based semantics. The idea is to prove matching logic properties on programs in the source language, and guarantee, via the compiler's correctness that the compiled programs in the target language satisfy those properties as well.

References

[1] Dorel Lucanu and Vlad Rusu. Program Equivalence by Circular Reasoning. In *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, Turku, Finland, June 2013. Springer.

[2] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A Generic Framework for Symbolic Execution. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301, Indianapolis, États-Unis, October 2013. Springer Verlag.

[3] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Languages Design and Implementation*, pages 327–337, 2009.

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

[5] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 2012. 10.1002/stvr.1472.

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

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

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

[9] 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.

[10] 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 *Computer-Aided Verification*, pages 185–198, 2005.

[11] Sorin Craciunescu. Proving the equivalence of CLP programs. In *International Conference of Logic Programming, LNCS 2401*, pages 287–301, 2002.

[12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification, LNCS 7358*, pages 712–717, 2012.

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

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

[15] Santiago Escobar and José Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.

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

[17] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer, 2007.

[18] Davide Ancona and Elena Zucca. Corecursive featherweight java. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 3–10, New York, NY, USA, 2012. ACM.

[19] Grigore Roşu 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.

[20] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A Generic Approach to Symbolic Execution. Research Report RR-8189, INRIA, 2012. `http://hal.inria.fr/hal-00766220/`.

[21] 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 *Proc. 13th Brazilian Symposium on Formal Methods, LNCS 6527*, pages 226–241, 2011.

[22] 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.

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

RESEARCH CENTRE
LILLE – NORD EUROPE

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

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

ISSN 0249-6399