Operationally-based Program Equivalence Proofs using LCTRSs

Ștefan Ciobâcă Dorel Lucanu Andrei Sebastian Buruiană
Alexandru Ioan Cuza University (Iași, Romania)
stefan.ciobaca@gmail.com dorel.lucanu@gmail.com sebastian.buruiana@yahoo.com

We propose an operationally-based framework for deductive proofs of program equivalence. It is based on encoding the language semantics as logically constrained term rewriting systems (LCTRSs) and the two programs as terms. Our method requires an extension of standard LCTRSs with axiomatized symbols. We also present a prototype implementation that proves the feasibility our approach.

1 Introduction: Language Semantics as LCTRSs

An LCTRS consists of rewrite rules of the form $l \longrightarrow r$ **if** ϕ, where l, r are terms and ϕ is a first-order constraint. The reduction relation $\longrightarrow$ induced by an LCTRS can define the transition relation modeling the operational semantics of programming languages: $P \longrightarrow Q$ iff the program configuration P *steps* into program configuration Q.

We feature a running example with an imperative language `WH`, where l and r are terms of sort Cfg representing program configurations. The language `WH` features global variables, boolean expressions, assignments, conditionals, while loops and first-order functions. In Figure 1 we present the syntax of WH in a BNF-like notation.

Exp	::=	**Int** | **Bool** | **Id** | **Exp** `binop` **Exp** | `unop` **Exp** | **call FunCall** | **skip** | **Exp**;**Exp** | **Id**:=**Exp** | **while Exp do Exp** | **if Exp then Exp else Exp**	
FunCall	::=	**Id** | **FunCall**(**Exp**)	**FunBody** ::= **Exp** | λ**Id**.**FunBody**
Stack	::=	[] | **Exp** ⇝ **Stack**	**Cfg** ::= ⟨**Stack**, **Env**, **Funcs**⟩
Env	::=	**Array**{**Int**}{**Int**}	**Funcs** ::= **Array**{**Id**}{**FunBody**}

Figure 1: The syntax of `WH` written in BNF-like notation. Each non-terminal is a sort. Subsorting is implemented in a many-sorted setting by an invisible injection from the smaller sort into the larger sort.

Values. The `WH` language has two types of values (expressions that have been completely evaluated): integers and booleans. The predicate **val** : **Exp** → **Bool** holds for expressions that are values. The constants of sort **Int** are $\ldots, -\mathbf{2}, -\mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots$. The constants of sort **Bool** are $\top$ and $\bot$.

Expressions and statements. There is no syntactic difference in the language between expressions and statements; both are grouped under the sort **Exp**. Any value is an expression. Identifiers (program variables), which are terms of sort **Id**, are also expressions. Binary (summation, multiplication, logical and, the relation less-than and others) and unary operands (unary minus, boolean negation, and possibly others) are represented by the constructs **Exp** `binop` **Exp** and `unop` **Exp** (`binop` ranges over $\{+, \leq, \ldots\}$ and `unop` ranges over $-$, **not**, ...). The special expression $\square$ (pronounced *hole*) is not part of the user-facing syntax and it is only used as an auxiliary construct by the rewriting rules defining the operational semantics.

Submitted to:
WPTE 2021

$\langle \mathsf{x}{:=}\mathsf{e} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e} \rightsquigarrow \mathsf{x}{:=}\Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e) *assignment*
$\langle \mathsf{i} \rightsquigarrow \mathsf{x}{:=}\Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{x}{:=}\mathsf{i} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle \mathsf{x}{:=}\mathsf{i} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{es},$ **update**$(\mathsf{env},\mathsf{x},\mathsf{i}),\mathsf{fs}\rangle$

$\langle \mathsf{x} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **lookup**$(\mathsf{x},\mathsf{env}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *identifier lookup*

$\langle \mathsf{e}_1 + \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_1 \rightsquigarrow \Box + \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e_1) *binary*
$\langle \mathsf{i}_1 \rightsquigarrow \Box + \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{i}_1 + \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *operations*
$\langle \mathsf{i}_1 + \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_2 \rightsquigarrow \mathsf{i}_1 + \Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e_2)
$\langle \mathsf{i}_2 \rightsquigarrow \mathsf{i}_1 + \Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{i}_1 + \mathsf{i}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle \mathsf{i}_1 + \mathsf{i}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{i}_1 + \mathsf{i}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$

$\langle$ **not** $\mathsf{e} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e} \rightsquigarrow$ **not** $\Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e) *unary*
$\langle \mathsf{b} \rightsquigarrow$ **not** $\Box \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **not** $\mathsf{b} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *operations*
$\langle$ **not** $\mathsf{b} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \overline{\mathsf{b}} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$

$\langle$ **if** $\top$ **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *if-then-else*
$\langle$ **if** $\bot$ **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle$ **if** e_1 **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_1 \rightsquigarrow$ **if** $\Box$ **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e_1)
$\langle \mathsf{b} \rightsquigarrow$ **if** $\Box$ **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **if** b **then** e_2 **else** $\mathsf{e}_3 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$

$\langle$ **while** e_1 **do** $\mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow$ *while loop*
$\quad \langle$ **if** e_1 **then** $(\mathsf{e}_2;$ **while** e_1 **do** $\mathsf{e}_2)$ **else skip** $\rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$

$\langle \mathsf{e}_1;\mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e}_1 \rightsquigarrow \mathsf{e}_2 \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *sequence*

$\langle$ **skip** $\rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *skip*

$\langle$ **call** $\mathsf{f} \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **lookup**$(\mathsf{f},\mathsf{fs}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ *function calls*
$\langle$ **call** $\mathsf{f}(\mathsf{e}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **call** $\mathsf{f} \rightsquigarrow$ **call** $\Box(\mathsf{e}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle \lambda\mathsf{x}.\mathsf{fb} \rightsquigarrow$ **call** $\Box(\mathsf{e}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\mathsf{e}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\mathsf{e}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle \mathsf{e} \rightsquigarrow$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\Box) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$ **if** $\neg$**val**(e)
$\langle \mathsf{i} \rightsquigarrow$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\Box) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\mathsf{i}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$
$\langle$ **call** $\lambda\mathsf{x}.\mathsf{fb}(\mathsf{i}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle \longrightarrow \langle$ **subst**$(\mathsf{x},\mathsf{i},\mathsf{fb}) \rightsquigarrow \mathsf{es},\mathsf{env},\mathsf{fs}\rangle$

Figure 2: The semantics of the WH language as a set of constrained rewrite rules.

The statements **skip**, **Exp**;**Exp** and **Id**:=**Exp** represent a no-op, sequential composition and assignment to a global variable, respectively. The statement **while Exp do Exp** represents the standard *while* loop. The construct **if Exp then Exp else Exp** represents conditional statements and expressions.

Program configuration. The semantics of a program is a transition system over *(program) configurations*. Configurations in WH are tuples $\langle [e_1,\ldots,e_n], env, fs\rangle$ consisting of: • a cons-list $[e_1,\ldots,e_n]$ of expressions and statements to be evaluated in order, representing the evaluation stack and • an environment *env* mapping identifiers to their values, • and an environment *fs* mapping function names to their bodies. We will use a Haskell-like notation for lists: $[]$ is the empty list, $\rightsquigarrow$ is the (right-associative) list constructor, and $[e_1,e_2,\ldots,e_n]$ is a shorthand for $e_1 \rightsquigarrow e_2 \rightsquigarrow \ldots \rightsquigarrow e_n \rightsquigarrow []$. The operational semantics of WH is given by the logically constrained rewrite rules in Figure 2. We use the following conventions: • standard font is used for meta-variables (e.g., l,r standing for terms and ϕ for constraints), • sans−serif, red font for object-level variables (e.g., e : **Exp**, i : **Int**) and • **bold**, **blue font** for object-level non-variable symbols (e.g., the function symbol **lookup** : **Env** × **Id** → **Int** or the sort **Env**).

The first three rules define the semantics of the assignment statements and we explain them in more

detail. The first rule ensures that if an assignment $x{:=}e$ is at the top of the stack ($x{:=}e \rightsquigarrow es$) and e is not a value (i.e., not an integer), then e is *scheduled* for evaluation, by placing it in front of the computation stack ($e \rightsquigarrow x{:=}\Box \rightsquigarrow es$), where the other semantic rules ensure its eventual evaluation to an integer. The special constant $\Box$ (pronounced *hole*) is used as a placeholder to recall which part of the statement has been promoted for evaluation. The second rule ensures that once e (in the stack $e \rightsquigarrow x{:=}\Box \rightsquigarrow es$) has been transformed into an integer i (the stack becomes $i \rightsquigarrow x{:=}\Box \rightsquigarrow es$) using the rules for evaluation of expressions, then the assignment statement is reconstructed: $x{:=}i \rightsquigarrow es$. The third rule defines the semantics of assigning an integer $i : \mathbf{Int}$ to the program identifier $x : \mathbf{Id}$. In this case, the environment is updated by using the **update** function (**update** is an interpreted function that is defined by the usual theory of arrays).

To evaluate an WH program expression e (a term of sort **Exp**), we start with the initial configuration $\langle e \rightsquigarrow [], env\rangle fs$, where env is some arbitrary environment and fs is the set of functions defined in the program and apply the constrained rewrite rules that define the semantics of WH until reaching a configuration that has no successor. This style of giving an operational semantics to a language is called *frame stack style* [24], was extended and popularized by the K framework [28] and it avoids the necessity of refocusing [11] typical of operational semantics based on evaluation contexts.

2 Contribution

We propose a method for proving equivalence of two programs written in two arbitrary languages whose semantics are given as LCTRSs. The method is based on two-sided simulation. The main advantage is that our method is parametric in the two semantics. This allows for easy experimentation with various settings encoded in the language semantics. We exploit this advantage to prove *equivalence in the presence of resource bounds:* it is easy to modify the operational semantics of WH to account for a bounded stack, by adding a constraint to each rewrite rule:

$$\langle x{:=}e \rightsquigarrow es, env, fs\rangle \longrightarrow \langle e \rightsquigarrow x{:=}\Box \rightsquigarrow es, env, fs\rangle \ \mathbf{if}\ \neg\mathbf{val}(e) \wedge \underbrace{\mathbf{len}(es) < k}_{\textit{new constraint}}$$

The new constraint imposes a bound of $k+1$ elements on the stack. We call the unbounded-stack version of the WH language $\mathtt{WH}_1$ and the bounded-stack version $\mathtt{WH}_2$; both have the same syntax, but slightly different semantics. We study using our method the equivalence in $\mathtt{WH}_1$ and $\mathtt{WH}_2$ of the programs

$$\langle [\mathbf{call}\ \mathbf{f}(N)], env, fs\rangle \qquad \text{and}$$
$$\langle [\mathbf{call}\ \mathbf{F}(N, \mathbf{0}, \mathbf{0})], env, fs\rangle$$

for $N \geq \mathbf{0}$, where the recursive functions **f** and **F** are defined by the function map

$$fs = \{\ \mathbf{f} \mapsto \lambda\mathbf{n}.\,\mathbf{if}\,\mathbf{n} = \mathbf{0}\,\mathbf{then}\,\mathbf{0}\,\mathbf{else}\,\mathbf{n} + \mathbf{call}\ \mathbf{f}(\mathbf{n}-\mathbf{1}),$$
$$\mathbf{F} \mapsto \lambda\mathbf{n}.\lambda\mathbf{i}.\lambda\mathbf{a}.\,\mathbf{if}\,\mathbf{i} \leq \mathbf{n}\,\mathbf{then}\,\mathbf{call}\ \mathbf{F}(\mathbf{n}, \mathbf{i}+\mathbf{1}, \mathbf{a}+\mathbf{i})\,\mathbf{else}\,\mathbf{a}\}.$$

Both functions compute the sum $1 + \ldots + n$ when the argument N is assigned a non-negative integer n and are therefore functionally equivalent. However, the function **f** uses $O(n)$ stack space, while **F** uses constant stack space (it is a tail-recursive version of **f**). Therefore there is an observable difference between **f** and **F** in $\mathtt{WH}_2$: **f** will generate a stack overflow for a sufficiently large input, while **F** will terminate succesfully. Our method can show that they are equivalent in $\mathtt{WH}_1$ (unbounded stack), but it (correctly) fails to prove them equivalent in $\mathtt{WH}_2$ (bounded stack).

To formally define the simulation relation, we require a technical improvement of LCTRSs in the form of *axiomatized symbols*. An axiomatized symbol is defined by a ground convergent LCTRS. In our example, we axiomatize a symbol **reduce** by

$$\mathbf{reduce}(\mathsf{i},\mathsf{n}) \longrightarrow [] \ \mathbf{if}\ \mathsf{i}{>}\mathsf{n},$$
$$\mathbf{reduce}(\mathsf{i},\mathsf{n}) \longrightarrow (\mathsf{i}{+}\Box) \rightsquigarrow \mathbf{reduce}(\mathsf{i}{+}\mathbf{1},\mathsf{n}) \ \mathbf{if}\ \mathsf{i}{\leq}\mathsf{n}.$$

Then $\mathbf{reduce}(\mathsf{i},\mathsf{n})$ stands for the informally presented cons-list $[\mathsf{i}+\Box,(\mathsf{i}+\mathbf{1})+\Box,\ldots,\mathsf{n}+\Box]$, which is used to match configurations in executions of **f** against corresponding configurations in executions of **F**. The axiomatized symbol **reduce** is used in Section 4 to define a simulation relation.

Contributions. • We propose an operationally-based framework for proving program equivalence that is more flexible than existing approaches. We illustrate this by a new application: equivalence proofs in the presence of resource bounds (e.g., stack size) • We extend our previous work on LCTRSs [8] by *axiomatized symbols*, which are critical for expressing powerful simulation relations. We identify a new problem in rewriting, *unification modulo axiomatized symbols*, a particular type of higher-order unificationt that appears naturally in the context of program equivalence, but might also be useful in other types of symbolic computation. • We implement the proof method in the prototype RMT tool; it can prove equivalences that are out of the reach of other verifiers.

Structure. In Section 3, we propose definitions and proof systems for partial and full equivalence. Section 4 presents the application, equivalence checking in the presence of resource bounds. We briefly survey related work in Section 5 before concluding in Section 6. The accompanying technical report [9] contains more details and applications of our method.

3 Simulation-based Equivalence Proofs

We assume that $\mathscr{R}_L$ and $\mathscr{R}_R$ are two LCTRSs modeling the semantics of two programming languages. Let CfgL and CfgR be the sorts of the corresponding configurations. The two languages can be the same or they can be different; all our results are parametric in $\mathscr{R}_L$ and $\mathscr{R}_R$. Formally, we define equivalence not between programs, but between *program configurations*. We sometimes distinguish between *symbolic program configurations* (terms of sort CfgL or CfgR, possibly with variables) and *ground program configurations* (elements of the interpretation of the sorts CfgL and CfgR). Our proof method shows equivalence between two symbolic program configurations. The fact that the same variable occurs in both symbolic configurations models that the two programs take the same input.

Sometimes two programs are equivalent, but end up in slightly different configurations. For example, an imperative program might store its result in a global variable **result**, while a functional program would simply reduce to its final value. We still want to be able to consider these programs equivalent. Therefore, we parameterize our definition for equivalence by a set of *base cases*, which define the pairs of terminal ground configurations that are considered to be equivalent. We denote by $\mathbb{B}$ (for base) the set of pairs of ground terminal program configurations that are known to be equivalent.

We propose two definitions for the notion of functional equivalence of programs, based on two-way simulations. In the following definitions, by a complete path $\rho(P) \longrightarrow^*_{\mathscr{R}_L} \mathrm{P}'$, we mean that no further reduction step is possible for P'.

Definition 1 (Full (Partial) Simulation) *A symbolic program configuration P* is fully (partially) simulated by a symbolic program configuration Q under constraint ϕ with a set of base cases $\mathbb{B}$, *written* $\mathbb{B} \vDash P \prec Q$ **if** ϕ ($\mathbb{B} \vDash P \preceq Q$ **if** ϕ) *if, for any valuation ρ such that $\rho(\phi) = \top$ and for any complete path $\rho(P) \longrightarrow^*_{\mathscr{R}_L} \mathrm{P}'$, there exists a complete path $\rho(Q) \longrightarrow^*_{\mathscr{R}_R} \mathrm{Q}'$ such that $(\mathrm{P}',\mathrm{Q}') \in \mathbb{B}$ (or – for partial simulation only – there exists an infinite path $\rho(Q) \longrightarrow_{\mathscr{R}_R} \ldots$);*

In **full** simulation, for any terminating run of the left hand side on some input, there is a terminating run of the right hand side on the same input, such that the results are part of $\mathbb{B}$ (e.g., the results are equal). For **partial** simulation, a terminating run of the lhs can be simulated by an infinite run of the right hand

side on the same input. The notion of *full (partial) simulation* is inspired from the usual notion of *full (partial) equivalence* [17] in the relational program verification literature. It can be seen as a lopsided version of full (partial) equivalence. Full simulation is a transitive relation (assuming the base cases are defined transitively). Partial simulation is not transitive (even for consistently defined sets of base cases). Note that $\prec \subseteq \preceq$ (for a fixed $\mathbb{B}$), which justifies the notation.

Definition 2 (Full (Partial) Equivalence) *Two symbolic program configurations P and Q are* fully equivalent (partially equivalent) *under constraint ϕ with a set of base cases $\mathbb{B}$, written $\mathbb{B} \vDash P \sim Q$ if ϕ ($\mathbb{B} \vDash P \simeq Q$ if ϕ,), if $\mathbb{B} \vDash P \prec Q$ if ϕ and $\mathbb{B}^{-1} \vDash Q \prec P$ if ϕ ($\mathbb{B} \vDash P \preceq Q$ if ϕ and $\mathbb{B}^{-1} \vDash Q \preceq P$ if ϕ).*

Two-way full (partial) simulation gives the usual notion of full (partial) equivalence for determinate programs. Partial equivalence is not an equivalence relation (hence the name *partial*). The notation is justified by $\sim \subseteq \simeq$ (for a fixed $\mathbb{B}$).

Proving full and partial simulation. We work with simulation formulae of the form $P \prec Q$ if ϕ for full simulation and of the form $P \preceq Q$ if ϕ for partial simulation, where P is a symbolic configuration of sort CfgL, Q is a symbolic configuration of sort CfgR and ϕ is a first-order logical constraint. To save space, we write $\dot{\preceq}$ for $\prec$ or $\preceq$ in contexts where both options are allowed. For a set R of formulae $P \dot{\preceq} Q$ if ϕ, its denotation is $[\![R]\!] = \{(\rho(P), \rho(Q)) \mid (P \dot{\preceq} Q \text{ if } \phi) \in R \text{ and } \rho \vDash \phi\}$ (i.e., the pairs of instances of P and Q that satisfy ϕ).

We fix a set B of simulation formulae that under-approximates [1] the set $\mathbb{B}$ of base cases: $[\![B]\!] \subseteq \mathbb{B}$. We also consider a set G (for *goals*) consisting of simulation formulae to be proven. The set G usually includes the actual goal, but also a set of intermediate helper goals that are needed for the proof that we call *circularities*. The proof systems for full and partial simulation, presented in Figure 3, manipulate sequents of the form $G, B \vdash^g P \prec Q$ if ϕ (for full simulation) and $G, B \vdash^g P \preceq Q$ if ϕ (for partial simulation), where $g \in \{0,1\}$; G is the set of goals and B is the set of base cases. The superscript g to the turnstile is a boolean flag (representing a *guard*) that denotes: 1. for full simulation: whether circularities are enabled or not, as formalized in the proof rules; 2. for partial simulation: $g = 1$ allows the CIRC$^{\preceq}$ rule to not make progress on the rhs. The approximate informal meaning of a sequent is $G, B \vdash^g P \prec Q$ if ϕ is that P simulates Q under the constraint ϕ with the set of base cases B if the simulations in G hold (see Theorem 1 for the exact meaning).

The AXIOM rule states that any P is simulated by any Q under the constraint $\bot$ (false). The BASE rule handles the case when the right hand side Q can take a number of steps into Q' so that the base cases are reached (the pair (P, Q') is part of the base cases). The constraint $sub\Big((P,Q),R\Big) \triangleq \bigvee_{P' \dot{\preceq} Q' \text{ if } \phi' \in R} \exists var(P',Q',\phi').(\phi' \wedge P{=}P' \wedge Q{=}Q')$ expresses that $P \dot{\preceq} Q$ is an instance of R by enumerating all sequents $P' \dot{\preceq} Q'$ if $\phi' \in R$ and collecting a constraint $\exists var(P',Q',\phi').(\phi' \wedge P{=}P' \wedge Q{=}Q')$ that intuitively captures the fact that $P \dot{\preceq} Q$ is an instance of the sequent. The only rule that is different between partial simulation and full simulation is CIRC. The CIRC rule handles the case when Q reaches in a number of steps a configuration Q' such that the pair P, Q' is subsumed by some circularity in G. For soundness, in full simulation, CIRC$^{\prec}$ can only be applied when the superscript for the turnstile, g, is 1. The superscript becomes 1 only in rule STEP, which intuitively takes a step in the left-hand side. Therefore, when the rule CIRC$^{\prec}$ is actually used, it means that there was progress on the lhs (by a *previous* STEP on the lhs). The rule CIRC$^{\preceq}$ is similar, but progress is required on the **rhs**, unless $g = 1$. Therefore, for partial simulation, it is allowed to discharge a goal directly by CIRC$^{\preceq}$, without taking any step in the

[1] In practice, B can usually be chosen such that its denotation matches exact the set of base cases: $[\![B]\!] = \mathbb{B}$. However, our proof system is still sound when B is an under-approximation.

Notation: $sub\Big((P,Q),R\Big) \triangleq \bigvee_{P' \dot{\preceq} Q' \textbf{ if } \phi' \in R} \exists var(P',Q',\phi').(\phi' \wedge P{=}P' \wedge Q{=}Q')$

$$\textsc{Axiom}\ \frac{}{G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \bot}$$

$$\textsc{Base}\ \frac{G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \phi \wedge \neg\phi_B}{G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \phi} \models \phi_B \rightarrow \bigvee_{Q' \textbf{ if } \phi' \in \Delta^{\leq k}_{\mathscr{R}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),B\Big)$$

$$\textsc{Circ}^{\prec}\ \frac{G,B \vdash^1 P \prec Q \textbf{ if } \phi \wedge \neg\phi_G}{G,B \vdash^1 P \prec Q \textbf{ if } \phi} \models \phi_G \rightarrow \bigvee_{Q' \textbf{ if } \phi' \in \Delta^{\leq k}_{\mathscr{R}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),G\Big)$$

$$\textsc{Circ}^{\preceq}\ \frac{G,B \vdash^g P \preceq Q \textbf{ if } \phi \wedge \neg\phi_G}{G,B \vdash^g P \preceq Q \textbf{ if } \phi} \models \phi_G \rightarrow \bigvee_{Q' \textbf{ if } \phi' \in \Delta^{\geq 1-g,\leq k}_{\mathscr{R}_R}(Q)} \phi' \rightarrow sub\Big((P,Q'),G\Big)$$

$$\textsc{Step}\ \frac{\begin{array}{l} G,B \vdash^1 P^i \dot{\preceq} Q \textbf{ if } \phi^i \text{ (for all } 1 \leq i \leq n) \\ G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \phi \wedge \neg\phi^1 \wedge \ldots \wedge \neg\phi^n \end{array}}{G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \phi} \Delta_{\mathscr{R}_L}(P \textbf{ if } \phi) = \{P^i \textbf{ if } \phi^i \mid 1 \leq i \leq n\}$$

Figure 3: The proof systems for full simulation and for partial simulation. The shorthand $\dot{\preceq}$ should be replaced consistently by either $\prec$ or $\preceq$ in any given rule.

left hand side, but with progress on the rhs (note the superscript $\geq 1-g$ to Δ). This corresponds to the case where a terminal configuration is partially simulated by an infinite loop. Another difference is that once rule STEP is applied to make progress on the lhs, circularities can be applied even if there is no progress on the rhs. This corresponds potentially to the case where the left-hand side loops forever and the right hand side finishes.

Finally, rule STEP can be used to take a symbolic step in the left-hand side. The set Δ computes the symbolic successors of a configuration w.r.t. a rewrite rule or rewrite system: $\Delta_{l,r,\phi}(t) = \{\sigma'(r) \textbf{ if } \phi' \wedge \phi \mid (\phi',\sigma') \in mgu(t,l), (\phi' \wedge \phi) \text{ satisfiable}\}$, where *mgu* computes a complete set of unifiers modulo axiomatized symbols and builtins. Note that all possible symbolic steps from P are taken. This corresponds to the fact that in our notion of simulation, *every* run of P must have a corresponding run in Q. The constraint $\neg\phi^1 \wedge \ldots \wedge \neg\phi^n$ describes the instances of P where no step can be taken, and therefore these configurations must be solved by some other rule, hence the second line in the hypotheses of STEP. Rules BASE and CIRC* have a condition, $\models \ldots$, which is implemented by an oracle (in practice, an SMT solver) deciding validity in a given theory. We write $G,B \vdash^g G'$ if $G,B \vdash^g P \dot{\preceq} Q \textbf{ if } \phi$ for any formula $P \dot{\preceq} Q \textbf{ if } \phi \in G'$ (i.e., all formulae in G' are provable from G,B). We are now ready to give the main soundness theorem of our result for full simulation and partial simulation.

Theorem 1 (Soundness for full/partial simulation) *If $G,B \vdash^0 G$ and $[\![B]\!] \subseteq \mathbb{B}$, then for any simulation formula $P \dot{\preceq} Q \textbf{ if } \phi \in G$, we have that $\mathbb{B} \models P \dot{\preceq} Q \textbf{ if } \phi$. The shorthand $\dot{\preceq}$ should be consistently replaced by either $\prec$ or $\preceq$.*

The theorem requires that *all formulae* in G be proved in order to trust *any* of them. If a formula in G is not provable then even if the others are provable, they cannot be trusted to hold semantically. The starting superscript of the turnstile must be 0, for soundness of CIRC. The conditions of the CIRC rules are designed to use goals in G as axioms safely, while keeping soundness.

4 Equivalence in the Presence of Resource Bounds

We continue the discussion in Section 2 and we show how our proof systems can be used to reason about equivalence in the presence of a bound on the stack. For the following examples, we choose $\mathscr{R}_L = \mathscr{R}_R =$ WH, CfgL = CfgR = **Cfg** and we show the equivalence of the symbolic program configurations

$$\langle [\textbf{call f}(\mathsf{N})], \mathsf{env}, fs\rangle \text{ and } \langle [\textbf{call F}(\mathsf{N}, \textbf{0}, \textbf{0})], \mathsf{env}, fs\rangle \text{ for } \mathsf{N} \geq \textbf{0}, \text{ where}$$

$fs = \{\, \textbf{f} \mapsto \lambda \textbf{n}.\, \textbf{if n} = \textbf{0 then 0 else n} + \textbf{call f}(\textbf{n} - \textbf{1}),\ \textbf{F} \mapsto \lambda \textbf{n}.\lambda \textbf{i}.\lambda \textbf{a}.\, \textbf{if i} \leq \textbf{n then call F}(\textbf{n}, \textbf{i}+\textbf{1}, \textbf{a}+\textbf{i})\ \textbf{else a}\}$ is the map containing the function bodies corresponding to the function identifiers **f** and **F**. We use $\textbf{F}(\mathsf{N}, \textbf{0}, \textbf{0})$ as an abbreviation for the syntactic construct $\textbf{F}(\mathsf{N})(\textbf{0})(\textbf{0})$ of sort **FunCall**. Note that N is a variable of sort **Int** and env is a variable of sort **Env** (map from program identifiers to integers). The fact that both N and env occur in the two symbolic configurations models the fact that we want the two programs to take the same input N and to start in the same environment env. In the initial configuration, the helper arguments of **F** are fixed to **0**.

Choice of base cases. We let $\mathbb{B} = \{(\langle [i], env, fs\rangle, \langle [i'], env', fs'\rangle) \mid i = i' \wedge i, i' \in \mathbb{Z}\}$. This means that we consider any terminal configurations that have reduced to the same integer $i = i'$ to be equivalent. We under-approximate $\mathbb{B}$ by the following set $B = \{\langle [\mathsf{i}], \mathsf{env}, \mathsf{fs}\rangle \dot{\preceq} \langle [\mathsf{i}'], \mathsf{env}', \mathsf{fs}'\rangle \textbf{ if } \mathsf{i} = \mathsf{i}'\}$ of formulae for the base cases (we happen to have a perfect approximation, as $[\![B]\!] = \mathbb{B}$). *Choice of goals.* The set of goals includes the actual goal to be proven and two helper circularities: $G = \{$

$$\begin{array}{lcll} \langle [\textbf{call f}(\mathsf{N})], \mathsf{env}, fs\rangle & \dot{\preceq} & \langle [\textbf{call F}(\mathsf{N}, \textbf{0}, \textbf{0})], \mathsf{env}, fs\rangle & \textbf{if} \quad \textbf{0} \leq \mathsf{N}, \\ \langle \textbf{call f}(\mathsf{I}-\textbf{1}) \rightsquigarrow \mathit{reduce}(\mathsf{I}, \mathsf{N}), \mathsf{env}, fs\rangle & \dot{\preceq} & \langle [\textbf{call F}(\mathsf{N}, \textbf{0}, \textbf{0})], \mathsf{env}, fs\rangle & \textbf{if} \quad \textbf{0} \leq \mathsf{I} \leq \mathsf{N}, \\ \langle \mathsf{S} \rightsquigarrow \mathit{reduce}(\mathsf{I}, \mathsf{N}), \mathsf{env}, fs\rangle & \dot{\preceq} & \langle [\textbf{call F}(\mathsf{N}, \mathsf{I}, \mathsf{S})], \mathsf{env}, fs\rangle & \textbf{if} \quad \textbf{1} \leq \mathsf{I} \leq \mathsf{N}\}, \end{array}$$

where fs is the previously defined function map, **n**, **i**, **a**, **f**, **F** are constants of sort **Id** (program identifiers), and where N, I, S : **Int** are variables.

Using the sets G, B defined above, we have that $G, B \vdash^0 G$ for partial and full simulation and that $G^{-1}, B^{-1} \vdash^0 G^{-1}$ for partial simulation. Our proof system cannot show $G^{-1}, B^{-1} \vdash^0 G^{-1}$ in the sense of full simulation, intuitively because it cannot prove that the termination of **F** (one phase) implies the termination of **f** (two phases). The full simulation relation would require an operationally-based termination argument [5] for the second phase of **f**, which we leave for future work. Next, we abbreviate $\langle [\textbf{call f}(\mathsf{N})], \mathsf{env}, fs\rangle$ by **f** and $\langle [\textbf{call F}(\mathsf{N}, \textbf{0}, \textbf{0})], \mathsf{env}, fs\rangle$ by **F**. We have used our implementation to show the following simulations. • We show that $\textbf{f} \prec \textbf{F}$, $\textbf{f} \preceq \textbf{F}$ and that $\textbf{F} \preceq \textbf{f}$ under the constraint $\mathsf{N} \geq \textbf{0}$ for our running example in the language $\mathtt{WH}_1$. As explained above, our method cannot show $\textbf{F} \prec \textbf{f}$ **if** $\mathsf{N} \geq \textbf{0}$. • In $\mathtt{WH}_2$, none of $\textbf{f} \prec \textbf{F}$, $\textbf{f} \preceq \textbf{F}$, $\textbf{F} \preceq \textbf{f}$, $\textbf{F} \preceq \textbf{f}$ hold under the constraint $\mathsf{N} \geq \textbf{0}$, and therefore these goals (correctly) fail. • Our method can also prove programs in two different languages as well. We show that **f**, interpreted in $\mathtt{WH}_1$, is partially equivalent to **F**, interpreted in $\mathtt{WH}_2$, when $\mathsf{N} \geq \textbf{0}$. For one direction ($\textbf{f} \prec \textbf{F}$), we establish full simulation; for the other direction, just partial simulation. • We show that: • a `while` loop is partially equivalent to a recursive function, when both compute the sum of the first N naturals, and • full equivalence holds for an instance of loop unswitching. This proves that our approach can handle structurally different programs.

Implementation. We have a prototype implementation of the two proof systems in the `RMT` tool at:

`http://profs.info.uaic.ro/~stefan.ciobaca/wpte2021`.

In addition to the bounded-stack example above, we have also used our tool to prove equivalence of several equivalence examples in the literature. Figure 4 summarizes our results on some equivalence benchmarks of Ctrl [16][2] and PEC [20].

[2]Examples found at `http://cl-informatik.uibk.ac.at/software/ctrl/tocl/`.

Example	Ctrl	RMT
fib02	4.02s	25.73s
fib03	3.06s	27.55s
fib04	Timeout	77.40s
fib05	3.99s	36.67s
fib06	32.81s	63.39s
fib07	2.74s	59.31s
fib08	3.44s	60.77s
fib09	3.09s	41.51s
fib10	2.34s	35.91s
fib11	Timeout	79.52s
fib12	No	No
sum01	2.39s	10.51s
sum02	2.33s	12.36s
sum03	2.62s	12.30s

(a)

Optimization	PEC	CORK	RMT
Code hoisting	✓	0.32s	0.41s
Constant propagation	✓	0.33s	0.31s
Copy propagation	✓	0.33s	0.26s
If-conversion	✓	0.34s	0.48s
Partial redundancy elimination	✓	0.34s	0.75s
Loop invariant code motion	✓	3.48s	3.79s
Loop peeling	✓	3.26s	0.97s
Loop unrolling	✓	12.17s	7.09s
Loop unswitching	✓	8.19s	4.71s
Software pipelining	✓	8.02s	3.56s
Loop fission	✓	23.45s	∗ 10.40s
Loop fusion	✓	23.34s	∗ 9.67s
Loop interchange	✓	29.30s	∗ 108.63s
Loop reversal	✓	8.41s	2.70s
Loop skewing	✓	8.50s	7.68s
Loop flattening	×	×	∗ 8.14s
Loop strength reduction	×	5.63s	5.26s
Loop tiling 01	×	10.94s	25.41s
Loop tiling 02			∗ 21.58s

(b)

Figure 4: In Subfigure a we compare against CTRL on a selection of examples [16]. Our tool RMT is slower, but it can handle some new examples. In Subfigure b we compare against PEC [20] and CORK [21] on a benchmark of equivalence proofs arrising in program optimizations. The annotation $*$ means that there is a technical difference in the definition of equivalence between RMT and CORK/PEC.

5 Related Work

Due to space limits, we only present a selection of relevant related work. A more extensive comparison can be found in the accompanying technical report [9]. Pitts [23] proposed the use of operationally-based notions of contextual equivalence and the *frame stack* approach [24] for small-step semantics that we use. The same style of using a frame stack was popularized by the K framework [28]. Logical relations and bisimulation can be used to prove contextual equivalence. Bisimulation techniques [27] are usually language dependent and proofs of congruence and other properties need to be established independently. Logical relations techniques [13] can be used to prove contextual equivalences for various languages. Several relational Hoare logics were proposed [3, 1] for reasoning about pairs of programs. A concept close to relational Hoare logic is that of product-program [2], which are programs that mimic the behavior of two programs; they allow to reduce relational reasoning to reasoning about a single program. Grimm et al. [18] propose a general method for relational proofs based on encoding the state transformation as a monad in the F* proof assistant. Kundu et al. [20] propose an implementation of a parametrized equivalence prover and we compare against the tool in Figure 4. Chaki et al. [6] propose a definition of equivalence suitable for nondeterministic programs, and introduce sound proof rules for regression verification of multithreaded programs. A technique for automated discovery of simulation relations is proposed by Fedyukovich et al. [15]. Techniques based on an efficient encoding of the

relational property as a set of constrained Horn clause are also possible [12]. A technique for automatic proving of equivalences for procedural programs based on LCTRSs is proposed by Fuhs et al. [16]. The main difference is that in our approach, the operational semantics of the two programs are also given as input. Logically constrained term rewriting systems, which combine term rewriting and SMT constraints are introduced by Kop et al. [19]. Rewriting modulo SMT is introduced by Rocha et al. [26] for analyzing open systems. *Our own related work.* We first considered semantics-based equivalence [22] for symbolic programs in the context of the K framework [28], but for a notion of behavioural equivalence of deterministic programs. We [10] gave a semantics-based proof system for full equivalence. Most of the infrastructure required for LCTRSs is based on our earlier work on proving reachability in LCTRSs [8] and solving unification modulo builtins [7].

6 Conclusion and Future Work

We have introduced and implemented in `RMT` a new method for proving simulation and equivalence in languages whose semantics are defined by LCTRSs in frame stack style. To express simulation relations, we enrich standard LCTRSs with *axiomatized symbols*, which raise new research questions. We also generalize existing definitions for full/partial equivalence. Our approach allows for nondeterminism in the definitions and proofs of equivalence, which we will exploit to its full potential in future work. We show the advantages of our framework in proving equivalence in the presence of resource bounds. The applications critically relies on easily changing the operational semantics of the language(s).

As future work, we would like to apply our methods to more challenging concurrent programs and to realistic rewrite-based language definitions, available as part of the K framework [14, 4]. We would also like to integrate an external termination checker to handle full equivalence better. Other directions for future work include relational cost analysis [25].

References

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg & Pierre-Yves Strub (2017): *A Relational Logic for Higher-order Programs*. *Proc. ACM Program. Lang.* 1(ICFP), pp. 21:1–21:29.

[2] Gilles Barthe, Juan Manuel Crespo & César Kunz (2016): *Product programs and relational program logics*. *Journal of Logical and Algebraic Methods in Programming* 85(5, Part 2), pp. 847 – 859.

[3] Nick Benton (2004): *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, ACM, New York, NY, USA, pp. 14–25.

[4] Denis Bogdănaş & Grigore Roşu (2015): *K-Java: A Complete Semantics of Java*. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, ACM, New York, NY, USA, pp. 445–456.

[5] Andrei-Sebastian Buruiană & Ştefan Ciobâcă (2019): *Reducing Total Correctness to Partial Correctness by a Transformation of the Language Semantics*. *CoRR* abs/1902.08419. Available at `http://arxiv.org/abs/1902.08419`.

[6] Sagar Chaki, Arie Gurfinkel & Ofer Strichman (2015): *Regression Verification for Multi-threaded Programs (with Extensions to Locks and Dynamic Thread Creation)*. *Form. Methods Syst. Des.* 47(3), pp. 287–301.

[7] Ştefan Ciobâcă, Andrei Arusoaie & Dorel Lucanu (2018): *Unification Modulo Builtins*. In: *Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018, Bogota, Colombia, July 24-27, 2018, Proceedings*, pp. 179–195, doi:10.1007/978-3-662-57669-4_10. Available at `https://doi.org/10.1007/978-3-662-57669-4_10`.

[8] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Oxford, UK, July 14-17, 2018, Proceedings*, pp. 295–311, doi:10.1007/978-3-319-94205-6_20. Available at `https://doi.org/10.1007/978-3-319-94205-6_20`.

[9] Ştefan Ciobâcă, Dorel Lucanu & Andrei Sebastian Buruiană (2020): *Operationally-based Program Equivalence Proofs using LCTRSs*. Technical Report submission id: 3018955, arXiv.

[10] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu & Grigore Roşu (2016): *A Language-independent Proof System for Full Program Equivalence*. *Form. Asp. Comput.* 28(3), pp. 469–497.

[11] Olivier Danvy & Lasse R. Nielsen (2004): *Refocusing in Reduction Semantics*. Technical Report RS-04-26, BRICS, Department of Computer Science, University of Aarhus.

[12] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2016): *Relational Verification Through Horn Clause Transformation*. In Xavier Rival, editor: *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 147–169.

[13] Derek Dreyer, Amal Ahmed & Lars Birkedal (2011): *Logical Step-Indexed Logical Relations*. *Logical Methods in Computer Science* 7(2), doi:10.2168/LMCS-7(2:16)2011. Available at `https://doi.org/10.2168/LMCS-7(2:16)2011`.

[14] Chucky Ellison & Grigore Roşu (2012): *An Executable Formal Semantics of C with Applications*. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, ACM, New York, NY, USA, pp. 533–544.

[15] Grigory Fedyukovich, Arie Gurfinkel & Natasha Sharygina (2015): *Automated Discovery of Simulation Between Programs*. In: *The 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning - Volume 9450*, LPAR-20 2015, Springer-Verlag, Berlin, Heidelberg, pp. 606–621.

[16] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Logic* 18(2), pp. 14:1–14:50.

[17] Benny Godlin & Ofer Strichman (2010): *Inference Rules for Proving the Equivalence of Recursive Procedures*. In Zohar Manna & Doron A. Peled, editors: *Time for Verification, Essays in Memory of Amir Pnueli*, *Lecture Notes in Computer Science* 6200, Springer, pp. 167–184, doi:10.1007/978-3-642-13754-9_8. Available at `https://doi.org/10.1007/978-3-642-13754-9_8`.

[18] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy & Santiago Zanella-Béguelin (2018): *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, ACM, New York, NY, USA, pp. 130–145.

[19] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[20] Sudipta Kundu, Zachary Tatlock & Sorin Lerner (2009): *Proving Optimizations Correct Using Parameterized Program Equivalence*. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM, New York, NY, USA, pp. 327–337.

[21] Nuno P. Lopes & José Monteiro (2016): *Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic*. *STTT* 18(4), pp. 359–374, doi:10.1007/s10009-015-0366-1. Available at `https://doi.org/10.1007/s10009-015-0366-1`.

[22] Dorel Lucanu & Vlad Rusu (2015): *Program Equivalence by Circular Reasoning*. *Form. Asp. Comput.* 27(4), pp. 701–726.

[23] A. M. Pitts (1996): *Reasoning About Local Variables with Operationally-based Logical Relations*. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, IEEE Computer Society, Washington, DC, USA, pp. 152–.

[24] Andrew M. Pitts (2002): *Operational Semantics and Program Equivalence*. In: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, Springer-Verlag, London, UK, UK, pp. 378–412.

[25] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg & Florian Zuleger (2017): *Monadic Refinements for Relational Cost Analysis*. *Proc. ACM Program. Lang.* 2(POPL), pp. 36:1–36:32.

[26] Camilo Rocha, José Meseguer & César Muñoz (2017): *Rewriting modulo SMT and open system analysis*. *Journal of Logical and Algebraic Methods in Programming* 86(1), pp. 269 – 297.

[27] Davide Sangiorgi, Naoki Kobayashi & Eijiro Sumii (2011): *Environmental Bisimulations for Higher-order Languages*. *ACM Trans. Program. Lang. Syst.* 33(1), pp. 5:1–5:69.

[28] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li & Grigore Roşu (2016): *Semantics-based Program Verifiers for All Languages*. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, ACM, New York, NY, USA, pp. 74–91.