Reducing Total Correctness to Partial Correctness by a Transformation of the Language Semantics

Andrei-Sebastian Buruiană
Alexandru Ioan Cuza University & Bitdefender
sburuiana@bitdefender.com

Ștefan Ciobâcă (✉)
Alexandru Ioan Cuza University
stefan.ciobaca@info.uaic.ro

We give a language-parametric solution to the problem of *total correctness*, by automatically reducing it to the problem of *partial correctness*, under the assumption that an expression whose value decreases with each program step in a well-founded order is provided. Our approach assumes that the programming language semantics is given as a rewrite theory. We implement a prototype on top of the RMT tool and we show that it works in practice on a number of examples.

1 Introduction

The line of work on reachability logic (see [22, 20, 21, 14, 15]) proposes language-parametric verification tools for programs. We continue this line of work by introducing a language-parametric total correctness checker. Our checker works by reducing the problem of total correctness to the problem of partial correctness by a transformation of the semantics of the programming language.

A program is *partially correct* if its output satisfies the postcondition for all inputs on which it terminates. A program is *totally correct* if it terminates on all inputs and its output satisfies the postcondition. Therefore, total correctness is usually proven by splitting the problem into two parts: first establish partial correctness by using various Hoare-like logics (e.g., [7, 15]), and then establish termination using a specialized termination prover (e.g., [17, 1]).

More rarely, logics that can directly prove total correctness (e.g., [25, 19]) are used. However, recent work in automated termination proving (e.g., [5, 3, 13, 6, 12]) shows that it is beneficial to use information obtained by proving properties of a program (e.g., invariants) in the termination argument. Most formal verification tools V take a (possibly annotated) program P as input and return $V(P)$, which is *yes* if the verification is successful and *no* if there is a counterexample; additionally, because such problems are typically undecidable, the verifier could return *unknown* or it could loop indefinitely. In this setting, if the programming language of P changes (e.g., when a new language standard is published), the verifier V needs to be upgraded as well and also proved sound – which may not be trivial. Another downside of this approach is that the same verification techniques need to be implemented and proved sound for all languages of interest.

In our line of work (see [22, 20, 21, 14, 15]), we propose to build language-parametric verifiers V: in this parametric setting, V takes as input both the (possibly annotated) program P and the operational semantics S of the programming language of P. Then $V(S,P)$ returns *yes*, *no* or *unknown* (or loops indefinitely), depending on the particular property that it checks of the program P in the operational semantics S. The advantage of this approach is that the verifier is proved sound once and can then be used for various programming languages.

Reachability logic, which is a sound and relatively complete proof system for partial correctness, was introduced in [14]. For a verifier V that implements this logic, $V(S,P)$ checks whether the (annotated) program P is partially correct, when interpreted using the operational semantics S. In the present

J. Niehren & D. Sabel (eds.): Rewriting Techniques
for Program Transformations and Evaluation (WPTE 2018)
EPTCS 289, 2019, pp. 1–16, doi:10.4204/EPTCS.289.1

$Id ::=$	$\mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \ldots$	*identifiers (program variables)*
$Int ::=$	$0, 1, -1, \ldots$	*integers*
$Bool ::=$	$True \mid False$	*booleans*
$AE ::=$	$Int \mid Id \mid AE + AE \mid \ldots$	*arithmetic expressions*
$BE ::=$	$Bool \mid AE = AE \mid AE < AE \mid not\ BE \mid \ldots$	*boolean expressions*
$Stmt ::=$	$skip$	*empty statement*
	$\mid Stmt; Stmt$	*sequence of statements*
	$\mid Id := AE$	*assignment*
	$\mid$ while BE do $Stmt$	*while loop*
	$\mid$ if BE then $Stmt$ else $Stmt$	*conditional statement*

Figure 1: The abstract syntax, in BNF-like notation, of the IMP language, which is used throughout the paper as a running example.

article, we propose to construct a language-parametric verifier $V_t(S,P)$ that checks *total correctness* of the program P in the operational semantics S. Our approach works by applying a transformation on S and the program P. We develop and prove the soundness of a transformation function θ such that $V_t(S,P) = V(\theta(S), \theta(P))$. This means that total correctness of the program P in the semantics S is the same as partial correctness of the program $\theta(P)$ in the semantics $\theta(S)$ and therefore the existing partial correctness verifier can be used in conjunction with the transformation θ to obtain a total correctness prover for any language.

Our approach assumes that the operational semantics S of the language in question is given as a rewrite theory with rules of the form

$$l \Rightarrow r \text{ if } b,$$

where l and r are two terms representing program configurations and b is a boolean constraint. For our running example, we use a simple imperative language that we call IMP (see, e.g., [26]), whose abstract syntax is presented in Figure 1. IMP configurations are pairs $\langle c_1 \rightsquigarrow c_2 \rightsquigarrow \cdots \rightsquigarrow c_n \rightsquigarrow Nil \mid env \rangle$ where $c_1, c_2, \ldots, c_n$ is a list of expressions or statements that are to be evaluated/executed in order and *env* is a map from program identifiers (program variables) to integers. The notation *Nil* stands for the empty list. The semantics of IMP consists of rewrite rules like

$$\begin{array}{lcll} \langle (v := i) \rightsquigarrow l \mid env \rangle & \Rightarrow & \langle l \mid update(v, i, env) \rangle & \text{and} \\ \langle (\text{if } b \text{ then } s_1 \text{ else } s_2) \rightsquigarrow l \mid env \rangle & \Rightarrow & \langle s_1 \rightsquigarrow l \mid env \rangle \text{ if } b = True, & \end{array}$$

which define the meaning of all language operators. The two rules above illustrate parts of the semantics of the assignment statement and of the if-then-else statement, respectively. The full details on the syntax and semantics of IMP are formally given in Section 2. However, we note that it is possible to faithfully model a variety of languages in this manner, as shown in [24]. Given a language semantics S as a parameter, reachability logic (defined in [14]) can prove sequents of the form

$$S \vdash l \wedge \phi_l \Rightarrow^{\forall} \exists \tilde{x}.(r \wedge \phi_r),$$

where l and r are configuration terms and ϕ_l, ϕ_r are constraints. The intuitive meaning of a sequent is that any instance of the configuration l satisfying constraint ϕ_l either diverges (does not terminate) or it reaches (along any path, hence the $\forall$) in a finite number of steps an instance of the configuration r satisfying constraint ϕ_r and agreeing with l on all variables except $\tilde{x}$. The full syntax and semantics of

the sequents are presented formally in Section 2. Note that such sequents subsume the notion of partial correctness. For example, the partial correctness of the SUM program

```
s := 0
while not (m = 0) do s := s + m; m := m - 1
```

is represented by the following partial correctness sequent

$$S \vdash \langle SUM \mid env_1 \rangle \wedge lookup(\mathtt{m}, env_1) = z \wedge z \geq 0 \Rightarrow^{\forall} \\ \exists env_2.(\langle skip \mid env_2 \rangle \wedge lookup(\mathtt{s}, env_2) = z(z+1)/2),$$

which is derivable using reachability logic. The sequent states that if we run the *SUM* program in a configuration where the environment env_1 maps the program identifier `m` to a positive integer z, then the program eventually reaches a configuration where there is nothing left to execute (hence the *skip*) and where the identifier `s` is mapped to the sum of the first z positive naturals. The sequent

$$S \vdash \langle SUM \mid env_1 \rangle \wedge lookup(\mathtt{m}, env_1) = z \Rightarrow^{\forall} \\ \exists env_2.(\langle skip \mid env_2 \rangle \wedge lookup(\mathtt{s}, env_2) = z(z+1)/2)$$

is also derivable (note that the constraint $z \geq 0$ does not appear anymore). The sequent is valid when interpreted in a partial correctness sense, since the program loops forever when $z < 0$. We propose a language transformation that builds an artificial semantics $\theta(S)$ from the semantics S by adding to the configuration a parameter that decreases with each rewrite step. The formal expression that is used for the parameter is a program variant (i.e., an expression whose value decreases with each program step). For example, the previously illustrated rewrite rules for the assignment statement and respectively for the conditional statement become:

$$\begin{array}{lcll} (\langle (v := i) \rightsquigarrow l \mid env \rangle, n) & \Rightarrow & (\langle l \mid update(v, i, env) \rangle, n-1) & \text{and} \\ (\langle (\text{if } b \text{ then } s_1 \text{ else } s_2) \rightsquigarrow l \mid env \rangle, n) & \Rightarrow & (\langle s_1 \rightsquigarrow l \mid env \rangle, n-1) \text{ if } b = True. & \end{array}$$

In the new semantics, $\theta(S)$, all programs terminate, since the variant is in a well-founded order and therefore it cannot decrease indefinitely. Therefore, in order to prove total correctness of a program P in S, it is sufficient to prove partial correctness of (P, B) in $\theta(S)$, where B is a sufficiently large bound. For our running example, we can establish that

$$\theta(S) \vdash (\langle SUM \mid env_1 \rangle, 200|z| + 200) \wedge lookup(\mathtt{m}, env_1) = z \wedge z \geq 0 \Rightarrow^{\forall} \\ \exists g, env_2.((\langle skip \mid env_2 \rangle, g) \wedge lookup(\mathtt{s}, env_2) = z(z+1)/2),$$

which implies by our soundness theorem that *SUM* is totally correct, under the precondition that the program variable `m` starts with a nonnegative value. We have chosen the upper bound $200|n| + 200$, since it is sufficiently large to allow for the program to finish. The variable g captures the number of execution steps remaining from the initial $200|n| + 200$ steps. The sequent above can be proven automatically (by relying on an invariant-like annotation for the while loop) in our implementation. However, by our soundness theorem, there is no bound B such that

$$\theta(S) \vdash (\langle SUM \mid env_1 \rangle, B) \wedge lookup(\mathtt{m}, env_1) = z \Rightarrow^{\forall} \\ \exists g, env_2.((\langle skip \mid env_2 \rangle, g) \wedge lookup(\mathtt{s}, env_2) = z(z+1)/2),$$

meaning that it is impossible to prove the total correctness of the program *SUM* if there is no precondition for the initial value of the program variable `m`.

In contrast with some other automated termination provers (discussed in Section 4), our method requires to provide the upper bound on the number of steps manually. In the example above, we picked $200|n|+200$ because, intuitively, the program has a linear-time complexity. The constant 200 should be large enough to allow the program to terminate.

The advantage and novelty of our method is that it is language-parametric (the semantics of the language is given as an input to our reduction). The main technical difficulties are to find a sound but general enough transformation θ (given in Definition 3.3) and the right statement of the soundness theorem (Theorem 3.1).

Contributions.

1. We propose a language-parametric method of proving total correctness;
2. Our approach works by reducing total correctness to partial correctness using a language transformation and therefore it can also be seen as an argument for semantics-parametric program verifiers;
3. We implement the reduction in the RMT [10] tool and we use it to prove several interesting examples.

Organization. In Section 2, we briefly introduce our notations for many-sorted algebras and we recall matching logic and reachability logic, which are the formalisms that we use to define and reason about the operational semantics of languages. In Section 3, we present our transformation, which reduces total correctness to partial correctness, we prove its soundness and we present the main difficulties. Section 4 discusses related work and Section 5 concludes the paper, including possible directions for future work.

2 Preliminaries: Proving Partial Correctness using Reachability Logic

This section fixes notations for many-sorted algebra and recalls matching logic and reachability logic. We denote by S^* the set of ordered tuples, possibly empty, with elements in S; we say that a set T is *S-indexed* if $T = \{T_s \mid s \in S\}$ is a collection of sets, each one corresponding to a different item in S. For ease of notation, we sometimes write $x \in T$ instead of $x \in T_s$ when s is clear from context. A *many-sorted signature* Σ is an ordered pair $\Sigma = (S,F)$, where S is the set of sorts, and $F = \{F_{w,s} \mid w \in S^*, s \in S\}$ is the $(S^* \times S)$-indexed set of function symbols. If $f \in F_{w,s}$, we say that f is a *function symbol* of arity (w,s). If $w = (s_1,\ldots,s_n)$, we sometimes write $f : s_1,\ldots,s_n \to s$ instead of $f \in F_{w,s}$, which indicates that the function symbol f has arguments of sorts $s_1,\ldots,s_n$ and a result of sort s ($f \in F_{w,s}$).

A Σ-algebra is a pair $\mathscr{A} = (A, I_A)$, where $A = \{A_s \mid s \in S\}$ is an S-indexed set called the carrier set of $\mathscr{A}$ and $I_A(f)$ is a function, $I_A(f) : A_{s_1} \times \ldots \times A_{s_n} \to A_s$, for all $f \in F_{(s_1,\ldots,s_n),s}$. That is, the interpretation map I_A assigns to each function symbol in F a function of the appropriate arity. For convenience, we sometimes refer to the algebra $\mathscr{A}$ as a set, in which case we mean its carrier set A. We assume as usual that $A_s \neq \emptyset$ for any $s \in S$. Given an S-indexed set of symbols *Var*, we denote by $Term_{\Sigma,s}(Var)$ the set of terms of sort s built with function symbols in Σ and variables in *Var* and by $Term_{\Sigma}(Var)$ the S-indexed set of all terms with variables in *Var*. Given a Σ-algebra $\mathscr{A}$ with carrier set $A = \{A_s \mid s \in S\}$, *a valuation* $\rho : \bigcup_{s\in S} Var_s \to \bigcup_{s\in S} A_s$ is a function that assigns to each variable an element in A of the appropriate sort. Valuations extend homomorphically to terms as usual. We now recall *matching logic*, as introduced in [14]. Fix an algebraic signature $\Sigma = (S,F)$ with a distinguished sort $Cfg \in S$ called the sort of configurations, an S-indexed set of variables *Var* and a Σ-algebra $\mathscr{T}$ with carrier set T. T is

$\overline{\cdot}$	:	$Int \to AE$	ε	:	$\to Env$
$[\cdot]$	:	$Id \to AE$	$\langle \cdot \mid \cdot \rangle$	:	$Stack \times Env \to Cfg$
$plus$	:	$AE \times AE \to AE$	$isInt$	:	$AE \to Bool$
$\underline{\cdot}$	:	$Bool \to BE$	$isBool$	:	$BE \to Bool$
eq	:	$AE \times AE \to BE$	$\cdot + \cdot$	:	$Int \times Int \to Int$
not	:	$BE \to BE$	$\cdot = \cdot$	:	$Int \times Int \to Bool$
$assign$	:	$Id \times AE \to Stmt$	$!\cdot$	:	$Bool \to Bool$
seq	:	$Stmt \times Stmt \to Stmt$	$lookup$	:	$Id \times Env \to Int$
ite	:	$BE \times Stmt \times Stmt \to Stmt$	$update$	:	$Id \times Int \times Env \to Env$
$while$	:	$BE \times Stmt \to Stmt$	$plushl$	:	$AE \to AE$
$skip$	:	$\to Stmt$	$plushr$	:	$AE \to AE$
$\llparenthesis \cdot \rrparenthesis$	:	$AE \to Code$	$eqhl$	:	$BE \to BE$
$[\![\cdot]\!]$	:	$Stmt \to Code$	$eqhr$	:	$BE \to BE$
$\lceil \cdot \rceil$	:	$BE \to Code$	$noth$	:	$\to BE$
Nil	:	$\to Stack$	$assignh$	:	$Id \to Stmt$
$\cdot \rightsquigarrow \cdot$	:	$Code \times Stack \to Stack$	$iteh$	:	$Stmt \times Stmt \to Stmt$

Figure 2: The symbols in the signature Σ used in our running example. For the infixed symbols, a centered dot represents an argument.

called the *configuration model*. The elements of the algebra $\mathscr{T}$ of sort *Cfg*, denoted by $\mathscr{T}_{Cfg}$, are called *configurations*. Matching logic is a logic of program configurations.

Example 2.1. We consider a running example where the elements of $\mathscr{T}$ of sort *Cfg* are programs, running in an environment, written in a simple imperative language that we call IMP. We work in the signature (S,Σ), where $S = \{Int, Bool, AE, BE, Id, Stmt, Stack, Env, Cfg, Code\}$ and where the function symbols in Σ are presented in Figure 2. The first set of symbols is used to represent the syntax of IMP programs. The second set of symbols is required to represent configurations, which consist of a stack of code to be executed/evaluated, and an environment mapping identifiers to integers. The third set of symbols represents mathematical operations. The last set consists of several auxiliary symbols, which are necessary to specify the rules of the operational semantics. For brevity, not all operators are presented; there are additional operations for less-than, boolean connectives, etc. The sorts *Int* and *Bool* are interpreted by mathematical integers and booleans, respectively. The sorts *AE*, *BE* and *Stmt* are the sorts for arithmetic expressions, boolean expressions and statements, respectively. The sort *Id* is for program identifiers (program variables). There are injections $\llparenthesis \cdot \rrparenthesis$, $\lceil \cdot \rceil$ and $[\![\cdot]\!]$ from *AE*, *BE* and *Stmt*, respectively, into the sort *Code*. Therefore *Code* refers to either arithmetic or boolean expressions, or statements. *Env* is the sort of maps from *Id*s to *Int*egers. The sort *Stack* refers to a stack of *Code*s that should be evaluated/executed in order, starting with the top of the stack. Configurations (of sort *Cfg*) consist of a *Stack* and of an environment of sort *Env*. The symbols in the signature Σ are presented in Figure 2. It includes all function symbols needed to represent the initial configuration, but also helper symbols that occur during program execution.

Example 2.2. The SUM program introduced earlier, placed in an initial configuration with the empty environment, ε, is represented by the following term of sort *Cfg*:

$$\langle [\![seq(assign(s,\overline{0}), \\ while(not(eq(\overline{0},[m])), seq(assign(s,plus([s],[m])), \\ assign(m,plus([m],\overline{-1})))))]\!] \rightsquigarrow Nil \mid \varepsilon \rangle.$$

The rest of this section recalls definitions from [14].

Definition 2.1. A *matching logic formula* (or *pattern*), is a first-order logic (FOL) formula that additionally allows terms in $Term_{\Sigma,Cfg}(Var)$, called *basic patterns*, as atomic formulae. We recall that by $Term_{\Sigma,Cfg}(Var)$ we denote the terms of sort *Cfg* in the Σ-algebra of terms. We say that a pattern is *structureless* if it contains no basic patterns. More formally, a matching logic formula is defined as follows:

1. if $\pi \in Term_{\Sigma,Cfg}(Var)$, then π is a formula;
2. if $w = (s_1,\ldots,s_n)$, $t_i \in Term_{\Sigma,s_i}(Var)$ for all $i \in \{1,\ldots,n\}$ and $P \in F_{w,Bool}$, then $P(t_1,\ldots,t_n)$ is a formula;
3. if φ_1 and φ_2 are formulae, then $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are formulae;
4. if φ is a formula, then $\neg\varphi$ is a formula;
5. if φ is a formula and $x \in Var$, then $\exists x\varphi$ and $\forall x\varphi$ are formulae.

By $\mathscr{P}_{\mathscr{T}}$ we denote the set of all patterns over an algebra $\mathscr{T}$.

Definition 2.2. For a fixed algebra $\mathscr{T} = (A, I)$, we define *satisfaction* $(\gamma,\rho) \models \varphi$ over configurations $\gamma \in \mathscr{T}_{Cfg}$, valuations $\rho : Var \to \mathscr{T}$ and patterns φ as follows:

1. $(\gamma,\rho) \models P(t_1,t_2,\ldots,t_n)$ if and only if $(I(P))(\rho(t_1),\rho(t_2),\ldots,\rho(t_n)) = \top$;
2. $(\gamma,\rho) \models \pi$ iff $\gamma = \rho(\pi)$ where $\pi \in Term_{\Sigma,Cfg}(Var)$;
3. $(\gamma,\rho) \models (\varphi_1 \wedge \varphi_2)$ iff $(\gamma,\rho) \models \varphi_1$ and $(\gamma,\rho) \models \varphi_2$;
4. $(\gamma,\rho) \models (\varphi_1 \vee \varphi_2)$ iff $(\gamma,\rho) \models \varphi_1$ or $(\gamma,\rho) \models \varphi_2$;
5. $(\gamma,\rho) \models \neg\varphi$ iff $(\gamma,\rho) \not\models \varphi$;
6. $(\gamma,\rho) \models \exists X\varphi$ iff $(\gamma,\rho') \models \varphi$ for some $\rho' : Var \to \mathscr{T}$ with $\rho'(y) = \rho(y)$ for all $y \in Var\backslash\{X\}$;
7. $(\gamma,\rho) \models \forall X\varphi$ iff $(\gamma,\rho) \not\models \exists X(\neg\varphi)$.

We write $\models \varphi$ when $(\gamma,\rho) \models \varphi$ for all $\gamma \in \mathscr{T}_{Cfg}$ and all $\rho : Var \to \mathscr{T}$.

We now recall all-path reachability logic (as presented in [14]).

Definition 2.3. A (one-path) *reachability rule* is an ordered pair of patterns (φ,φ') (which can have free variables). We write this pair as $\varphi \Rightarrow^{\exists} \varphi'$. We say that rule $\varphi \Rightarrow^{\exists} \varphi'$ is *weakly well-defined* iff for any $\gamma \in \mathscr{T}_{Cfg}$ and $\rho : Var \to \mathscr{T}$ with $(\gamma,\rho) \models \varphi$, there exists $\gamma' \in \mathscr{T}_{Cfg}$ such that $(\gamma',\rho) \models \varphi'$.

Definition 2.4. A *reachability system* is a set of reachability rules. A reachability system S is *weakly well-defined* iff each rule is weakly well-defined. S induces a *transition system* $(\mathscr{T}, \Rightarrow_S^{\mathscr{T}})$ on the configuration model: $\gamma \Rightarrow_S^{\mathscr{T}} \gamma'$ for $\gamma,\gamma' \in \mathscr{T}_{Cfg}$ iff there is some rule $\varphi \Rightarrow^{\exists} \varphi' \in S$ and some valuation $\rho : Var \to \mathscr{T}$ such that $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$. We write $\Rightarrow$ instead of $\Rightarrow_S^{\mathscr{T}}$ when it is clear from context that we are referring to a particular transition system.

Example 2.3. We consider a fixed Σ-algebra $\mathscr{T}$ having the following properties: $\mathscr{T}_{Int} = \mathbb{Z}$, $\mathscr{T}_{Bool} = \{True, False\}$, $\mathscr{T}_{Id} = \{x,y,z\ldots\}$, $\mathscr{T}_{a+b} = a+b$ *for all* $a,b \in \mathbb{Z}$, $\mathscr{T}_{lookup(X,update(X,I,env))} = I$ *for all* $X \in \mathscr{T}_{Id}, I \in \mathbb{Z}$, $env \in \mathscr{T}_{Env}$, $\mathscr{T}_{lookup(Y,update(X,I,env))} = \mathscr{T}_{lookup(Y,env)}$ *for all* $Y \in \mathscr{T}_{Id} \setminus \{X\}, I \in \mathbb{Z}$, $env \in \mathscr{T}_{Env}$, $\mathscr{T}_{lookup(X,\varepsilon)} = 0$ *for all* $X \in \mathscr{T}_{Id}$, $\mathscr{T}_{isInt(x)} = True$ iff $x = \overline{y}$, for some $y \in \mathbb{Z}$ and $\mathscr{T}_{isBool(x)} = True$ iff $x = \underline{y}$, for some $y \in \mathscr{T}_{Bool}$. The weakly well-defined system S defining the operational semantics of IMP is presented in Figure 3. For brevity, some rules that are similar to existing rules are missing (e.g., the rules for *eq* are similar to those for *plus*). We discuss the first four rules, which define the assignment operator and the lookup. The first rule schedules the expression on the rhs of an assignment to be evaluated, if it is

$$\langle [\![assign(X,A)]\!] \rightsquigarrow T \mid env\rangle \wedge \neg isInt(A) \Rightarrow^{\exists} \langle \llbracket A \rrbracket \rightsquigarrow [\![assignh(X)]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket \overline{a} \rrbracket \rightsquigarrow [\![assignh(X)]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle [\![assign(X,\overline{a})]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle [\![assign(X,\overline{a})]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle T \mid update(X,a,env)\rangle$$
$$\langle \llbracket [X] \rrbracket \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle \llbracket \overline{lookup(X,env)} \rrbracket \rightsquigarrow T \mid env\rangle$$
$$\langle [\![skip]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle T \mid env\rangle$$
$$\langle [\![seq(S_1,S_2)]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle [\![S_1]\!] \rightsquigarrow [\![S_2]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle [\![ite(\lceil \underline{False} \rceil, S_1, S_2)]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle [\![S_2]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle [\![ite(\lceil \underline{True} \rceil, S_1, S_2)]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle [\![S_1]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle [\![ite(C,S_1,S_2)]\!] \rightsquigarrow T \mid env\rangle \wedge \neg isBool(C) \Rightarrow^{\exists}$$
$$\quad \langle \lceil C \rceil \rightsquigarrow [\![iteh(S_1,S_2)]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle \lceil C \rceil \rightsquigarrow [\![iteh(S_1,S_2)]\!] \rightsquigarrow T \mid env\rangle \wedge isBool(C) \Rightarrow^{\exists}$$
$$\quad \langle [\![ite(C,S_1,S_2)]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle [\![while(C,S)]\!] \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle [\![ite(C,seq(S,while(C,S)),skip)]\!] \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket plus(\overline{a},\overline{b}) \rrbracket \rightsquigarrow T \mid env\rangle \Rightarrow^{\exists} \langle \llbracket \overline{a+b} \rrbracket \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket plus(A,B) \rrbracket \rightsquigarrow T \mid env\rangle \wedge \neg isInt(A) \Rightarrow^{\exists} \langle \llbracket A \rrbracket \rightsquigarrow \llbracket plushl(B) \rrbracket \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket plus(A,B) \rrbracket \rightsquigarrow T \mid env\rangle \wedge isInt(A) \wedge \neg isInt(B) \Rightarrow^{\exists}$$
$$\quad \langle \llbracket B \rrbracket \rightsquigarrow \llbracket plushr(A) \rrbracket \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket A \rrbracket \rightsquigarrow \llbracket plushl(B) \rrbracket \rightsquigarrow T \mid env\rangle \wedge isInt(A) \Rightarrow^{\exists} \langle \llbracket plus(A,B) \rrbracket \rightsquigarrow T \mid env\rangle$$
$$\langle \llbracket B \rrbracket \rightsquigarrow \llbracket plushr(A) \rrbracket \rightsquigarrow T \mid env\rangle \wedge isInt(B) \Rightarrow^{\exists} \langle \llbracket plus(A,B) \rrbracket \rightsquigarrow T \mid env\rangle$$

Figure 3: The reachability system S defining the semantics of IMP. Capital letters represent variables of the appropriate sorts. The variables a,b stand for integers and the variable *env* for an environment.

not already an integer. Once the expression is evaluated to an integer (using the other rules), the second rule places the result back into the assignment operator. Once the rhs is an integer, the third rule updates the environment appropriately. The fourth rule evaluates a variable by looking it up in the environment. The reachability system S generates the transition relation $\Rightarrow_S^{\mathscr{T}}$ on the model $\mathscr{T}$. Note that reachability rules of the form $l \wedge \phi \Rightarrow^{\exists} r$ (with $l \wedge \phi$ and r being matching logic formulae) subsume the rewrite rules of the form $l \Rightarrow r$ if φ used in the introduction.

Definition 2.5. A $\Rightarrow_S^{\mathscr{T}}$*-execution* is a sequence $\gamma_0 \Rightarrow_S^{\mathscr{T}} \gamma_1 \Rightarrow_S^{\mathscr{T}} \cdots$, potentially infinite, where $\gamma_0, \gamma_1, \ldots \in \mathscr{T}_{Cfg}$. If a $\Rightarrow_S^{\mathscr{T}}$-execution is finite, we call it a $\Rightarrow_S^{\mathscr{T}}$*-path*. We say that such a path is *complete* iff it is not a strict prefix of any other $\Rightarrow_S^{\mathscr{T}}$-path (i.e., the last element is irreducible).

The following is an example of a complete $\Rightarrow_S^{\mathscr{T}}$-path:

$$\langle [\![seq(skip,skip)]\!] \rightsquigarrow Nil \mid \varepsilon\rangle \Rightarrow \langle [\![skip]\!] \rightsquigarrow [\![skip]\!] \rightsquigarrow Nil \mid \varepsilon\rangle \Rightarrow \langle [\![skip]\!] \rightsquigarrow Nil \mid \varepsilon\rangle \Rightarrow \langle Nil \mid \varepsilon\rangle.$$

Definition 2.6 (Partial Correctness)**.** An *all-path reachability rule* is a pair $\varphi \Rightarrow^{\forall} \varphi'$. We say that $\varphi \Rightarrow^{\forall} \varphi'$ is *satisfied* by S, denoted by $S \models \varphi \Rightarrow^{\forall} \varphi'$, iff for all complete $\Rightarrow_S^{\mathscr{T}}$-paths τ starting with $\gamma \in \mathscr{T}_{Cfg}$ and for all $\rho : Var \to \mathscr{T}$ such that $(\gamma,\rho) \models \varphi$, there exists some $\gamma' \in \tau$ such that $(\gamma',\rho) \models \varphi'$.

The definition above generalizes typical partial correctness of Hoare tuples of the form $\{\varphi\}P\{\varphi'\}$, as the reachability formula $P \wedge \varphi \Rightarrow^{\forall} \langle skip \mid env\rangle \wedge \varphi'$ can be used instead. See [14] for a more detailed discussion. Reachability logic has a sound and relatively complete proof system, which derives sequents of the form $S \vdash \varphi \Rightarrow^{\forall} \varphi'$ if and only if $S \models \varphi \Rightarrow^{\forall} \varphi'$ holds. The results in the present paper do not depend on the proof system, and therefore the proof system is presented in Appendix A.

3 The Reduction of Total Correctness to Partial Correctness

We now present a transformation that reduces total correctness to the problem of partial correctness. We first define what it means for a pattern to terminate.

Definition 3.1 (Termination of a Pattern)**.** We say that a pattern φ *terminates* in S if for all $\gamma \in \mathscr{T}_{Cfg}$ and all $\rho : Var \to \mathscr{T}$ such that $(\gamma, \rho) \models \varphi$, all executions $\gamma \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots$ from γ in $(\mathscr{T}, \Rightarrow_S^{\mathscr{T}})$ are finite.

Example 3.1. The following pattern does *not* terminate in S:

$$\langle [\![while(C, skip)]\!] \rightsquigarrow Nil \mid \varepsilon \rangle, \text{ where } C \in Var_{BE}.$$

Its nontermination is witnessed by the following execution:

$$\begin{gathered} \langle [\![while([\![\underline{True}]\!], skip)]\!] \rightsquigarrow Nil \mid \varepsilon \rangle \Rightarrow_S^{\mathscr{T}} \\ \langle [\![ite([\![\underline{True}]\!], seq(skip, while([\![\underline{True}]\!], skip)), skip)]\!] \rightsquigarrow T \mid \varepsilon \rangle \Rightarrow_S^{\mathscr{T}} \cdots \\ \langle [\![while([\![\underline{True}]\!], skip)]\!] \rightsquigarrow Nil \mid \varepsilon \rangle \Rightarrow_S^{\mathscr{T}} \cdots \end{gathered}$$

The next definition is at the core of our proof. It is the total-correctness counterpart to Definition 2.6.

Definition 3.2 (Total Correctness)**.** We say that an all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ is *totally satisfied* by S, denoted by $S \models_t \varphi \Rightarrow^{\forall} \varphi'$, iff for all complete or diverging $\Rightarrow_S^{\mathscr{T}}$-executions τ starting with $\gamma \in \mathscr{T}_{Cfg}$ and for all $\rho : Var \to \mathscr{T}$ such that $(\gamma, \rho) \models \varphi$, there exists some $\gamma' \in \tau$ such that $(\gamma', \rho) \models \varphi'$.

We now discuss how the definition above generalizes the usual definition for total correctness found in the literature.

A Hoare tuple $\{\phi\} P \{\phi'\}$ is valid in the sense of total correctness if the precondition ϕ entails

1. the termination of the program P, and also
2. that the postcondition ϕ' holds after the program P terminates.

Our definition of $S \models_t \varphi \Rightarrow^{\forall} \varphi'$ states that any execution starting from φ, terminating or not, reaches at some point φ'. If we choose φ' to be a configuration that is known to terminate (e.g., for the case of IMP, $\langle skip \rightsquigarrow Nil \mid \ldots \rangle$), then it follows that φ must terminate along all paths. Otherwise, any nonterminating path starting with φ would meet φ', which terminates, leading to a contradiction.

In particular, the total correctness of the Hoare tuple $\{\phi\} P \{\phi'\}$ is encoded by $S \models_t P \wedge \phi \Rightarrow^{\forall} \langle skip \rightsquigarrow Nil \mid env \rangle \wedge \phi'$. In addition to encoding total correctness Hoare tuples, our definition of total correctness is strictly more general, since it guarantees that φ' is reached in a finite number of steps from φ, even if φ does not terminate.

We now present our transformation θ, which helps reduce total correctness guarantees of the form $S \models_t \varphi \Rightarrow^{\forall} \varphi'$ to partial correctness sequents of the form $\theta(S) \vdash \theta(\varphi, s) \Rightarrow^{\forall} \exists M . \theta(\varphi', M)$, where θ transforms its arguments as explained in Theorem 3.1 below.

Definition 3.3 (Reduction From Total Correctness to Partial Correctness)**.** We define several homonymous maps θ that encode our transformation for reducing total correctness to partial correctness. By $\mathscr{S}_\Sigma$ we denote the class of all algebraic signatures, by $\mathscr{S}$ the class of all sorts and by $\mathscr{U}$ the class of all algebras with distinguished sets of configurations.

1. *Transforming signatures* $(\theta : (\mathscr{S}_\Sigma \times \mathscr{S}) \to (\mathscr{S}_\Sigma \times \mathscr{S}))$
   Let $\Sigma = (S, F)$ be an algebraic signature and $Cfg \in S$. We define $\theta(\Sigma, Cfg) = (\Sigma', Cfg')$, where $\Sigma' = (S \cup Nat \cup Cfg', F \cup \left\{ F_{(),Nat}, F_{(Cfg,Nat),Cfg'}, F_{(Nat,Nat),Nat} \right\})$ and where $F_{(),Nat} = \{0, 1, 2, \ldots, \}$, $F_{(Cfg,Nat),Cfg'} = \{(,)\}$, $F_{(Nat,Nat),Nat} = \{+, -, \times, /\}$.

Intuitively, θ adds a sort for the set of naturals and changes the configuration sort such that new configurations consist of old configurations, plus a natural number. The natural intuitively represents a program variant that is added to the configuration, i.e. the maximum number of steps the program can take before ending its execution. In addition to the standard operations $+,-,\times,/$, we may also consider other operations like $|\cdot| : \mathit{Int} \to \mathit{Nat}$ (absolute value) that operate on *Nat* and other existing sorts. Alternatively, we could consider any well-founded set instead of the set of naturals; however, naturals make the presentation easier to follow.

2. *Transforming algebras* $(\theta : \mathscr{U} \to \mathscr{U})$

   Let $\mathscr{A} = (A, I_A)$ be a Σ-algebra, where *Cfg* is the distinguished sort of configurations and assume $\theta(\Sigma, \mathit{Cfg}) = (\Sigma', \mathit{Cfg}')$. Then $\theta(\mathscr{A}) = (A', I'_A)$ is a Σ'-algebra with a distinguished sort Cfg' defined as follows:

   (a) $A \subseteq A'$;
   (b) $\mathbb{N} = A'_{\mathit{Nat}} \in A'$;
   (c) I'_A is an extension of I_A such that $\mathscr{T}(n) = n_{\mathbb{N}}$, $\mathscr{T}(a \delta b) = \mathscr{T}(a \delta_{\mathbb{N}} b)$ for $\delta \in \{+,-,\times,/\}$;
   (d) $\mathscr{T}(\mathit{Cfg}') = \mathscr{T}(\mathit{Cfg}) \times \mathbb{N}$.

   Intuitively, each Σ-algebra with a distinguished sort *Cfg* of configurations is transformed into a Σ-algebra with a distinguished sort Cfg'. The sort Cfg' is interpreted as pairs of old configurations and naturals.

3. *Transforming patterns (matching logic formulae)* $(\theta : (\mathscr{P}_{\mathscr{T}} \times \mathit{Term}_{\Sigma,\mathit{Nat}}(\mathit{Var})) \to \mathscr{P}_{\theta(\mathscr{T})})$

   Consider a Σ-algebra $\mathscr{T}$. Let φ be a pattern over $\mathscr{T}$ and $n \in \mathit{Term}_{\Sigma,\mathit{Nat}}(\mathit{Var})$ (n is a term of sort *Nat*). We define θ as follows:

   (a) if φ is structureless, then $\theta(\varphi, n) = \varphi$;
   (b) if φ is a basic pattern, then $\theta(\varphi, n) = (\varphi, n)$ (note that this is the interesting case, as in the other cases the transformation θ simply applies homomorphically);
   (c) if $\varphi = (\varphi_1 \, \delta \, \varphi_2)$ and φ is not structureless, then $\theta(\varphi, n) = \theta(\varphi_1, n) \, \delta \, \theta(\varphi_2, n)$, for $\delta \in \{\vee, \wedge\}$;
   (d) if $\varphi = \delta X(\varphi')$ and φ is not structureless, then $\theta(\varphi, n) = \delta X \theta(\varphi', n)$, for $\delta \in \{\exists, \forall\}$;
   (e) if $\varphi = \neg\varphi'$ and φ is not structureless, then $\theta(\varphi, n) = \neg\theta(\varphi', n)$.

   Intuitively, θ transforms each old basic pattern into a new basic pattern by adding the natural n and "propagates" this change for all basic patterns contained in the given pattern.

4. *Transforming one-path reachability rules* $(\theta : (\mathscr{P}_{\mathscr{T}} \times \mathscr{P}_{\mathscr{T}}) \to (\mathscr{P}_{\theta(\mathscr{T})} \times \mathscr{P}_{\theta(\mathscr{T})}))$

   Let $\varphi \Rightarrow^{\exists} \varphi'$ be a reachability rule. Then $\theta(\varphi \Rightarrow^{\exists} \varphi') = \theta(\varphi, n) \Rightarrow^{\exists} \theta(\varphi', n-1)$, where n is a fresh variable of sort *Nat*. The transformation forces each rule to decrease the program variant (by 1).

5. *Transforming language semantics* $(\theta : 2^{(\mathscr{P}_{\mathscr{T}} \times \mathscr{P}_{\mathscr{T}})} \to 2^{(\mathscr{P}_{\theta(\mathscr{T})} \times \mathscr{P}_{\theta(\mathscr{T})})})$

   We define the transformation by $\theta(S) = \left\{ \theta(\varphi \Rightarrow^{\exists} \varphi') \mid (\varphi \Rightarrow^{\exists} \varphi') \in S \right\}$. Each one-path reachability rule is transformed independently.

We now reduce the problem of total correctness to partial correctness. This is achieved by the following property of the transformation θ defined previously:

Theorem 3.1. If there exists some term $s \in \mathit{Term}_{\Sigma,\mathit{Nat}}(\mathit{Var})$ of sort *Nat* such that

$$\theta(S) \models \theta(\varphi, s) \Rightarrow^{\forall} \exists M. \theta(\varphi', M),$$

where $M \in \mathit{Var}_{\mathit{Nat}}$, then $S \models_t \varphi \Rightarrow^{\forall} \varphi'$.

Proof. Suppose there exist some valuation $\rho : Var \to \theta(\mathscr{T})$, some configuration $\gamma \in \mathscr{T}_{Cfg}$ with the property $(\gamma,\rho) \models \varphi$ and a complete or diverging $\Rightarrow_S^{\mathscr{T}}$-execution $\tau = \gamma \Rightarrow_S^{\mathscr{T}} \gamma_1 \Rightarrow_S^{\mathscr{T}} \cdots$ such that there is no γ' in τ for which $(\gamma',\rho) \models \varphi'$. Let $n = \rho(s)$. As $\theta(S) \models \theta(\varphi,s) \Rightarrow^{\forall} \exists M.\theta(\varphi',M)$, we have, by definition, that for all complete $\Rightarrow_{\theta(S)}^{\theta(\mathscr{T})}$-paths $\tau^\theta = (\gamma^\theta,n) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_1^\theta,n-1) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} \cdots \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_k^\theta,n-k)$ such that $((\gamma^\theta,n),\rho) \models \theta(\varphi,s)$, there exists some $(\gamma_p^\theta,n-p)$ in τ^θ such that $((\gamma_p^\theta,n-p),\rho) \models \exists M.\theta(\varphi',M)$.

We distinguish two cases. First, suppose τ is complete and has at most n steps. Consider the path $\tau^\theta = (\gamma,n) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_1,n-1) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} \cdots \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_k,n-k)$. It is easy to see that since τ has at most n steps, τ^θ is indeed a valid $\Rightarrow_{\theta(S)}^{\theta(\mathscr{T})}$-path. Moreover, since τ is complete, it is easy to see that τ^θ is also complete. It follows that there exists some $(\gamma_p,n-p)$ in τ^θ such that $((\gamma_p,n-p),\rho) \models \exists M.\theta(\varphi',M)$. By the definition of satisfaction, this statement implies that $(\gamma_p,\rho) \models \varphi'$ and we have obtained a contradiction.

For the second case, we have that τ has more than n steps. Consider the prefix of τ of n steps: $\tau' = \gamma \Rightarrow_S^{\mathscr{T}} \gamma_1 \Rightarrow_S^{\mathscr{T}} \cdots \Rightarrow_S^{\mathscr{T}} \gamma_n$. Consider the $\Rightarrow_{\theta(S)}^{\theta(\mathscr{T})}$-path $\tau'' = (\gamma,n) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_1,n-1) \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} \cdots \Rightarrow_{\theta(S)}^{\theta(\mathscr{T})} (\gamma_n,0)$. Note that τ'' is indeed a valid path in $\theta(S)$ and additionally $((\gamma,n),\rho) \models \theta(\varphi,s)$. Moreover, τ'' is complete since $(\gamma_n,0)$ cannot advance in $\Rightarrow_{\theta(S)}^{\theta(\mathscr{T})}$.

This means that $((\gamma_p,n-p),\rho) \models \exists M.\theta(\varphi',M)$ for some value p. It is easy to see from the definition of satisfaction that this last statement implies $(\gamma_p,\rho) \models \varphi'$. Since $\theta(\gamma_p,0)$ is in τ'', then γ_p is in τ', which obviously implies that γ_p is in τ as well. Therefore, there exists γ_p in τ for which $(\gamma_p,\rho) \models \varphi'$.

We have arrived at a contradiction in both cases, from which we draw the conclusion that for all complete or diverging $\Rightarrow_S^{\mathscr{T}}$-paths τ starting with $\gamma \in \mathscr{T}_{Cfg}$ such that $(\gamma,\rho) \models \varphi$, there exists some γ' in τ such that $(\gamma',\rho) \models \varphi'$. By definition, this means that $S \models (\varphi \Rightarrow_t^{\forall} \varphi')$, which is what we had to prove. □

Corollary 3.1. If there exists $s \in Term_{\Sigma,Nat}(Var)$ of sort *Nat* such that $\theta(S) \models \theta(\varphi,s) \Rightarrow^{\forall} \exists M.\theta(\varphi',M)$, where $M \in Var_{Nat}$, then:

1. $S \models \varphi \Rightarrow^{\forall} \varphi'$;
2. If φ' terminates in S, then φ also terminates in S.

The converse of the corollary above, stating that if a partial correctness guarantee holds and φ terminates then the total correctness guarantee holds as well, in the cases of finitely-branching transition systems (this is an immediate consequence of König's lemma). Given the program SUM in our running example and the semantics S of IMP, the following sequent can be derived:

$$\theta(S) \vdash (\langle SUM \mid env_1 \rangle, 200|z|+200) \wedge lookup(m,env_1) = z \wedge z \geq 0 \Rightarrow^{\forall} \exists M, env_2.((\langle skip \mid env_2 \rangle, M) \wedge lookup(s,env_2) = z(z+1)/2),$$

which proves the total correctness of *SUM*. A fully worked out example of a proof of total correctness is given in Appendix B.

4 Related Work

We critically rely on previous work on language-parametric partial program correctness, as developed in [14]. Starting with the operational semantics of the language of the program for which we prove total correctness, we transform it into an (artificial) language whose configurations consist of the configurations of the initial language, plus a variant. This construction is automated. Given a program and a program variant, its total correctness in the original language reduces to showing partial correctness in

the new language. Language transformations have been used before, for example to develop language-parametric symbolic execution engines [18] or language-parametric partial equivalence checkers [9].

In general, the research community treats the subject of termination orthogonally to the subject of partial correctness. There are several automated approaches to proving (and certifying) termination (e.g., [17, 2, 11, 1]), but these are typically only concerned with termination, and not correctness. Therefore, to establish total correctness we generally first establish partial correctness by using various Hoare-like logics (e.g., [7, 15]), and then termination using a specialized termination prover (e.g., [17, 1]).

Logics that prove total correctness directly (e.g., [25, 19]) are used more rarely. This is despite the fact that relatively recent work in automated termination proving (e.g., [5, 3, 13, 6, 12]) shows that it is beneficial to use information obtained by proving a program (e.g., invariants) in the termination argument: in [5], a cooperation graph is used to enable the cooperation between a safety prover and the rank synthesis tool, in [3], a variance analysis is introduced that is parametric in an invariance analysis and Ramsey-based termination arguments are improved with lexicographic ordering in [13].

5 Conclusion and Future Work

We have developed a language semantics transformation that can be used to prove total correctness of programs. The method can be used for any programming language whose operational semantics is given by a set of reachability rules. This is not a restriction, as any programming language [24] can be faithfully encoded as such. Moreover, our definition of total correctness (Definition 3.2) generalizes the usual definition of total correctness, as it can also be used to reason about nonterminating programs that are guaranteed to reach a desired configuration (which could be nonterminating) in a finite number of steps. We have implemented our approach in the RMT tool [10, 8]. Instructions on obtaining RMT are available at at `http://profs.info.uaic.ro/~stefan.ciobaca/wpte2018`, along with several examples for total correctness (including our running example). Our examples show that our approach works in practice, but in future work we must also benchmark realistic languages with reachability logic semantics such as C (see [16]) or Java (see [4]). A limitation of our approach is that the number of steps has to be computable upfront. This means that we cannot handle programs that nondeterministically choose a value and loop for that number of steps. Another limitation is that the upper bound is not found automatically (even in simple cases), it has to be provided by the user.

There remain many exciting open questions for future work. The main question is proving our reduction to be complete. We will also study how our notion of total correctness corresponds to the well-known notions of may-convergence and must-convergence in the literature on process algebra (e.g.,in [23]). Another open question is whether our generalization of the notion of total correctness has any practical advantages over the usual definition. In our present approach, the program variant must be a natural number, but an important question is to analyze whether other well-founded orders could be needed as well. Another open question is compositionality: instead of providing a program-wide variant, would it be possible to have a more modular approach? Finally, can we combine our method with existing state of the art automated termination provers like [6, 12] to obtain the benefits of both?

Acknowledgement

This work is funded by the Ministry of Research and Innovation within Program 1 – Development of the national RD system, Subprogram 1.2 – Institutional Performance – RDI excellence funding projects, Contract no.34PFE/19.10.2018.

References

[1] Beatriz Alarcon, Raul Gutierrez, Jose Iborra & Salvador Lucas (2007): *Proving Termination of Context-Sensitive Rewriting with MU-TERM*. *ENTCS* 188, pp. 105 – 115, doi:10.1016/j.entcs.2007.05.041.

[2] Martin Avanzini, Christian Sternagel & René Thiemann (2015): *Certification of Complexity Proofs using CeTA*. In: *RTA*, *LIPIcs* 36, pp. 23–39, doi:10.4230/LIPIcs.RTA.2015.23. Available at http://drops.dagstuhl.de/opus/volltexte/2015/5187.

[3] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano & Peter O'Hearn (2007): *Variance Analyses from Invariance Analyses*. In: *POPL*, pp. 211–224, doi:10.1145/1190216.1190249.

[4] Denis Bogdănaş & Grigore Roşu (2015): *K-Java: A Complete Semantics of Java*. In: *POPL*, pp. 445–456, doi:10.1145/2676726.2676982.

[5] Marc Brockschmidt, Byron Cook & Carsten Fuhs (2013): *Better Termination Proving through Cooperation*. In: *CAV*, pp. 413–429, doi:10.1007/978-3-642-39799-8_28.

[6] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf & Nir Piterman (2016): *T2: Temporal Property Verification*. In: *TACAS*, pp. 387–393, doi:10.1007/978-3-662-49674-9_22.

[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds & Andrew W. Appel (2018): *VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs*. *JAR*, doi:10.1007/s10817-018-9457-5.

[8] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In: *IJCAR*, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[9] Ştefan Ciobâcă (2014): *Reducing Partial Equivalence to Partial Correctness*. In: *SYNASC*, pp. 164–171, doi:10.1109/SYNASC.2014.30.

[10] Ştefan Ciobâcă & Dorel Lucanu (2016): *RMT: Proving Reachability Properties in Constrained Term Rewriting Systems Modulo Theories*. Technical Report TR 16-01, Alexandru Ioan Cuza University, Faculty of Computer Science.

[11] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons & Xavier Urbain (2007): *Certification of Automated Termination Proofs*. In: *FroCoS*, pp. 148–162, doi:10.1007/978-3-540-74621-8_10.

[12] Byron Cook, Andreas Podelski & Andrey Rybalchenko (2006): *Termination Proofs for Systems Code*. In: *PLDI*, pp. 415–426, doi:10.1145/1133981.1134029.

[13] Byron Cook, Abigail See & Florian Zuleger (2013): *Ramsey vs. Lexicographic Termination Proving*. In: *TACAS*, pp. 47–61, doi:10.1007/978-3-642-36742-7_4.

[14] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă & Grigore Roşu (2014): *All-Path Reachability Logic*. In: *RTA-TLCA*, pp. 425–440, doi:10.1007/978-3-319-08918-8_29.

[15] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li & Grigore Roşu (2016): *Semantics-Based Program Verifiers for All Languages*. In: *OOPSLA*, pp. 74–91, doi:10.1145/2983990.2984027.

[16] Chucky Ellison & Grigore Roşu (2012): *An Executable Formal Semantics of C with Applications*. In: *POPL*, pp. 533–544, doi:10.1145/2103656.2103719.

[17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski & René Thiemann (2017): *Analyzing Program Termination and Complexity Automatically with AProVE*. *JAR* 58(1), pp. 3–31, doi:10.1007/s10817-016-9388-y.

[18] Dorel Lucanu, Vlad Rusu & Andrei Arusoaie (2017): *A generic framework for symbolic execution: a coinductive approach*. *J. Symb. Comput.* 80, pp. 125–163, doi:10.1016/j.jsc.2016.07.012.

[19] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner & Julian Sutherland (2016): *Modular Termination Verification for Non-Blocking Concurrency*. In: *ESOP*, pp. 176–201, doi:10.1007/978-3-662-49498-1_8.

[20] Grigore Roşu & Andrei Ştefănescu (2012): *Checking Reachability using Matching Logic*. In: *OOPSLA*, pp. 555–574, doi:10.1145/2384616.2384656.

[21] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă & Brandon M. Moore (2013): *One-Path Reachability Logic*. In: *LICS*, pp. 358–367, doi:10.1109/LICS.2013.42.

[22] Grigore Roşu, Chucky Ellison & Wolfram Schulte (2010): *Matching Logic: An Alternative to Hoare/Floyd Logic*. In: *AMAST*, *LNCS* 6486, pp. 142–162, doi:10.1007/978-3-642-17796-5_9.

[23] Manfred Schmidt-Schauß & David Sabel (2010): *Closures of may-, should-and must-convergences for contextual equivalence*. *Information Processing Letters* 110(6), pp. 232–235, doi:10.1016/j.ipl.2010.01.001.

[24] Traian Florin Şerbănuţă, Grigore Roşu & José Meseguer (2009): *A Rewriting Logic Approach to Operational Semantics*. *Information and Computation* 207(2), pp. 305–340, doi:10.1016/j.ic.2008.03.026.

[25] Dominic Steinhöfel & Nathan Wasser (2017): *A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows*. In: *iFM*, pp. 279–294, doi:10.1007/978-3-319-66845-1_18.

[26] Glynn Winskel (1993): *The formal semantics of programming languages. Foundations of Computing*.

$$\textsc{Step}\quad \frac{\models \varphi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in S} \exists \mathit{FreeVars}(\varphi_l)\varphi_l \qquad \models \exists c(\varphi[c/\Box] \wedge \varphi_l[c/\Box]) \wedge \varphi_r \rightarrow \varphi' \text{ for all } \varphi_l \Rightarrow^{\exists} \varphi_r \in S}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

$$\textsc{Axiom}\quad \frac{\varphi \Rightarrow^{\forall} \varphi' \in \mathscr{A}}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

$$\textsc{Transitivity}\quad \frac{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1 \Rightarrow^{\forall} \varphi_2 \qquad \mathscr{S}, \mathscr{A} \cup \mathscr{C} \vdash \varphi_2 \Rightarrow^{\forall} \varphi_3}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1 \Rightarrow^{\forall} \varphi_3}$$

$$\textsc{Case Analysis}\quad \frac{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1 \Rightarrow^{\forall} \varphi \qquad \mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_2 \Rightarrow^{\forall} \varphi}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1 \vee \varphi_2 \Rightarrow^{\forall} \varphi}$$

$$\textsc{Circularity}\quad \frac{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C} \cup \{\varphi \Rightarrow^{\forall} \varphi'\}} \varphi \Rightarrow^{\forall} \varphi'}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

$$\textsc{Abstraction}\quad \frac{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi \Rightarrow^{\forall} \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \exists X \varphi \Rightarrow^{\forall} \varphi'}$$

$$\textsc{Reflexivity}\quad \frac{\cdot}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi \Rightarrow^{\forall} \varphi}$$

$$\textsc{Consequence}\quad \frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1' \Rightarrow^{\forall} \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathscr{S}, \mathscr{A} \vdash_{\mathscr{C}} \varphi_1 \Rightarrow^{\forall} \varphi_2}$$

Figure 4: The language-parametric proof system for partial correctness in [14]

A Proof System for Partial Correctness

We recall in Figure 4 the proof system for the problem of partial correctness from [14].

Matching logic formulae can be translated into FOL formulae such that matching logic satisfaction reduces to FOL satisfaction in the model of configurations $\mathscr{T}$. This allows conventional theorem provers to be used for matching logic reasoning. One of the proof rules of reachability logic depends on this translation.

Definition A.1. Let $\Box$ be a fresh variable of sort *Cfg*. For a pattern φ, let $\varphi^{\Box}$ be the FOL formula formed from φ by replacing basic patterns $\pi \in \mathit{Term}_{\Sigma,\mathit{Cfg}}(\mathit{Var})$ with equalities $\Box = \pi$. If $\rho : \mathit{Var} \rightarrow \mathscr{T}$ and $\gamma \in \mathscr{T}_{\mathit{Cfg}}$ then let the valuation $\rho^{\gamma} : \mathit{Var} \cup \{\Box\}$ be such that $\rho^{\gamma}(x) = \rho(x)$ for all $x \in \mathit{Var}$ and $\rho^{\gamma}(\Box) = \gamma$.

We have that

$$(\gamma, \rho) \models \varphi \iff \rho^{\gamma} \models \varphi^{\Box}.$$

We use $\varphi[c/\Box]$ to denote the FOL formula resulting from eliminating $\Box$ from φ and replacing it with a *Cfg* variable c.

The proof system was shown in [14] to be sound (and also relatively complete) for the problem of partial correctness. Note that, this provides no guarantees for configurations that do not terminate.

B A Complete Example

In this section, we present in full details a very simple example of how the reduction presented above work. We consider a very simple "language" with configurations of the form $[s,i]$ (where s and i are naturals) that add to s the first i positive naturals.

Let $\Sigma = (\{Cfg, Nat, Bool\}, F)$, where:

- $F_{(),Nat} = \{0,1,2\ldots\}$
- $F_{(),Bool} = \{True, False\}$
- $F_{(Nat,Nat),Nat} = \{+,-,/,*\}$
- $F_{(Nat,Nat),Bool} = \{<,>,\leq,\geq,=\}$
- $F_{(Nat,Nat),Cfg} = \{[,]\}$

We consider a Σ-algebra $\mathscr{T}$ with the expected interpretation for common symbols and a system of reachability rules S consisting of a single rule:

$$[s,i] \wedge (i > 0) \Rightarrow [s+i, i-1], \text{ where } s,i \in Var_{Nat}.$$

The algebra $\theta(\mathscr{T})$ contains a sort Cfg' and, by definition, $\theta(S)$ consists of the following rule:

$$([s,i],n) \wedge (i > 0) \Rightarrow ([s+i, i-1], n-1), \text{ where } s,i,n \in Var_{Nat}.$$

For ease of readability let $SUM(x,y) = y*(y+1)/2-(x-1)*x/2$ by notation. Let $\varphi_L = ([SUM(n'+1,n),n'],n') \wedge n' \geq 0$ and $\varphi_R = \exists m([SUM(1,n),0],m)$, where $n',n,m \in Var_{Nat}$. Let us now prove that

$$\theta(S) \vdash ([0,n],n) \Rightarrow^{\forall} \varphi_R,$$

which establishes not only that $[0,n]$ computes the sum from 1 to n (by the soundness of reachability logic), but also that it terminates within n steps (by Theorem 3.1):

14. $\theta(S), \{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\} \vdash \exists n' \varphi_L \Rightarrow^{\forall} \varphi_R$ by **Axiom**
13. $\theta(S), \{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\} \vdash ([SUM(n',n),n'-1],n'-1) \wedge (n'-1) \geq 0 \Rightarrow^{\forall} \varphi_R$ by **Consequence** from 14
12. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} ([SUM(n'+1,n),n'],n') \wedge n' > 0 \Rightarrow^{\forall}$ $([SUM(n',n),n'-1],n'-1) \wedge (n'-1) \geq 0$ by **Step**
11. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} ([SUM(n'+1,n),n'],n') \wedge n' > 0 \Rightarrow^{\forall} \varphi_R$ by **Transitivity** from 12 and 13
10. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} ([SUM(1,n),0],0) \Rightarrow^{\forall} ([SUM(1,n),0],0)$ by **Reflexivity**
9. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} ([SUM(1,n),0],0) \Rightarrow^{\forall} \varphi_R$ by **Consequence** from 10
8. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} \exists n'(([SUM(n'+1,n),n'],n') \wedge n' > 0) \Rightarrow^{\forall} \varphi_R$ by **Abstraction** from 11
7. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} \exists (n'([SUM(n'+1,n),n'],n') \wedge n' = 0) \Rightarrow^{\forall} \varphi_R$ by **Consequence** from 9
6. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} \exists n'(([SUM(n'+1,n),n'],n') \wedge n' > 0) \vee$ $\exists n'(([SUM(n'+1,n),n'],n') \wedge n' = 0) \Rightarrow^{\forall} \varphi_R$ by **Case analysis** from 7, 8
5. $\theta(S) \vdash_{\{\exists n' \varphi_L \Rightarrow^{\forall} \varphi_R\}} \exists n' \varphi_L \Rightarrow^{\forall} \varphi_R$ by **Consequence** from 6

4. $\theta(S) \vdash ([0,n],n) \Rightarrow^{\forall} ([0,n],n)$ by **Reflexivity**
3. $\theta(S) \vdash \exists n' \varphi_L \Rightarrow^{\forall} \varphi_R$ by **Circularity** from 5
2. $\theta(S) \vdash ([0,n],n) \Rightarrow^{\forall} \exists n' \varphi_L$ by **Consequence** from 4
1. $\theta(S) \vdash ([0,n],n) \Rightarrow^{\forall} \varphi_R$ by **Transitivity** from 2 and 3

Our approach also works on the programming language IMP described above. We have shown, for example, that the following program is (unsurprisingly) totally correct (when `m` starts up with a nonnegative number):

```
s := 0
while not (m = 0) do s := s + m; m := m - 1
```

The main idea in proving the program above totally correct is the same as in the fully developed example above, but the formal proof is a lot longer.