Reducing Partial Equivalence to Partial Correctness

Ştefan Ciobâcă
Faculty of Computer Science
"Alexandru Ioan Cuza" University
Iaşi, Romania
stefan.ciobaca@gmail.com

***Abstract*—Two programs P and Q are partially equivalent if, when both terminate on the same input, they end up with equivalent outputs. Establishing partial equivalence is useful in, e.g., compiler verification, when P is the source program and Q is the target program, or in compiler optimization, when P is the initial program and Q is the optimized program.**

A program R is partially correct if, when it terminates, it ends up in a "good" state.

We show that, somewhat surprisingly, the problem of establishing partial equivalence can be reduced to the problem of showing partial correctness in an aggregated language, where programs R consist of pairs of programs $\langle P, Q\rangle$.

Our method is crucially based on the recently-introduced *matching logic*, which allows to faithfully define the operational semantics of any language. We show that we can construct the aggregated language mechanically, from the semantics of the initial languages. Furthermore, matching logic gives us for free a proof system for partial correctness for the resulting language. This proof system can then be used to prove partial equivalence.

I. Introduction

Starting with Hoare [1], it became obvious in the research community that proving critical programs to be correct (formal verification) is an important task in the software engineering lifecycle. Proving programs takes many shapes such as model checking or deductive proofs.

In this paper, we concentrate on a particular type of formal verification, the problem of establishing if two programs have the same behavior (i.e., if they are equivalent). There are multiple definitions for equivalence, depending on the application at hand. In this paper, we use the notion of partial equivalence. Two programs are partially equivalent if, when given the same input on which they both terminate, they end up with the same output.

This notion of equivalence is useful, for example, in establishing the correctness of compilers, where the compiled code needs to be equivalent to the source code, or of optimizations, where the optimized source code needs to have the same behavior as the unoptimized code.

We formally define the notion of partial equivalence, but instead of showing how to establish partial equivalence, we show a more powerful result: we show that proving partial equivalence of two programs can be reduced constructively to the problem of showing partial correctness of another program in a more complex language. This gives a method of establishing partial equivalence via a problem that is understood better (partial correctness), but also has some philosophical implications.

Our method is critically based on matching logic [2], [3], [4], [5], a logic that we describe in Section II and which provides us for free with a proof system for partial correctness [6], [7], [8], [9]. In Section III, we show how to build constructively the aggregated language in which partial correctness questions are equivalent to the initial partial equivalence questions. In Section IV we prove our main theorem that states that partial equivalence is reducible to partial correctness. We conclude our paper with Section V, in which we also mention related and further work.

This paper is supported by the European Sectorial Operational Programme Human Resource Development (SOP HRD), and by the Romanian Government under the contract number POSDRU/159/1.5/S/137750.

II. Preliminaries

Let S be a set of sorts with a distinguished sort $\mathit{Cfg} \in S$ and Σ an S-sorted signature. Let $\mathcal{T}$ be a Σ-model.

Definition 1. *The tuple* $(\mathit{Cfg}, S, \Sigma, \mathcal{T})$ *is called a* matching logic semantic domain.

If $s \in S$ is a sort, we denote by $\mathcal{T}_s$ the interpretation of the sort. In particular, $\mathcal{T}_{\mathit{Cfg}}$ is the set of configurations. The matching logic semantic domain defines the abstract syntax of the language (via the signature Σ) and the model of static configurations of the language (via the model $\mathcal{T}$). We next introduce as a running example the matching logic semantic domain for a simple imperative language that we call IMP.

Example 1. *The signature* Σ_{IMP} *is summarized in the following table as a grammar. Nonterminals represent sorts and productions represent function symbols:*

```
Int   ::= ... | -1 | 0 | 1 | ...
Var   ::= x | y | ...
ExpI  ::= Var | Int | ExpI op ExpI

Stmt  ::= Var := ExpI
        | skip | Stmt ; Stmt
        | if ExpI then Stmt else Stmt
        | while ExpI do Stmt
Code  ::= ExpI | Stmt
CfgI  ::= ⟨Code, Map{Var, Int}⟩
```

The operator `op` *ranges over* `+`, `*`, `/`, *etc. We discuss some of the productions above. The binary function symbol* `_;_` *takes as inputs two* `Stmts` *(statements) and constructs a new* `Stmt`*. The binary symbol* $\langle$`_, _`$\rangle$ *is used to create* I*MP configurations out of some code (denoted by the sort* `Code`*) and a heap (the memory of the program) of sort* `Map`$\{$`Var`, `Int`$\}$*. We denoted by* `Map`$\{$`Var`, `Int`$\}$ *the sort of the maps from* `Var` *to* `Int`*. We write such maps as comma-separated lists of tuples* $v \mapsto i$*, where the variable* v *is the key and the integer* i *is the value associated to* v *by the map.*

The model $\mathcal{T}$ *interprets the sort* `Int` *as the set of integer numbers, the set* `Var` *as the set of strings denoting variable names, all function symbols are interpreted freely and the sort* `Map`$\{$`Var`, `Int`$\}$ *is interpreted as actual maps between strings denoting variables names and integers.*

The sort `CfgI` *of* I*MP configurations is inhabited by pairs of code and heap. The code contains the code that remains to be executed, while the heap denotes the state of the memory of the program.*

If $s \in S$ is a sort, we denote by $T_s(Var)$ the set of terms of sort s over the variables *Var*. Given a matching logic semantic domain, *matching logic formulae* extend FOL formulae with terms $\pi \in T_{Cfg}(Var)$ of sort *Cfg* as atomic formulae:

Definition 2. *A* matching logic formula φ *is defined as follows:*

$$\varphi \doteq \exists x.\varphi, \neg\varphi, \varphi \wedge \varphi, \boxed{\pi}$$

In the above definition, x is a variable and the case in $\boxed{gray}$ is new when compared to FOL. The term $\pi \in T_{Cfg}(Var)$ used as an atomic formula is called a basic pattern.

Matching logic formulae are interpreted in the presence of valuations $\rho : Var \Rightarrow \mathcal{T}$, but also of an element $\gamma \in \mathcal{T}_{Cfg}$ of the domain:

Definition 3. *The* matching logic satisfaction *relation* $\models$ *(written* $(\gamma, \rho) \models \varphi$ *and read as* (γ, ρ) *is a model of* φ*) is defined inductively as follows:*

1) $(\gamma, \rho) \models \exists x.\varphi'$ *(where* x *has sort* $s \in S$*) iff there exists an element* $e \in \mathcal{T}_s$ *such that* $(\gamma, \rho[e/x]) \models \varphi'$ *(*$\rho[e/x]$ *is the valuation obtained from* ρ *by mapping* x *to* e*).*
2) $(\gamma, \rho) \models \neg\varphi$ *if* $(\gamma, \rho) \not\models \varphi$
3) $(\gamma, \rho) \models \varphi \wedge \varphi'$ *iff* $(\gamma, \rho) \models \varphi$ *and* $(\gamma, \rho) \models \varphi'$
4) $\boxed{(\gamma, \rho) \models \pi}$ *(for a basic pattern* $\pi \in T_{Cfg}(Var)$*) iff* $\boxed{\rho(\pi) = \gamma}$.

The part in $\boxed{gray\ background}$ is new when compared to FOL. The intuition behind the name "matching logic" is that the semantics of basic patterns π is given by matching against an element of the domain.

Example 2. *Let* $\varphi = \langle$`skip`$, x \mapsto x\rangle \wedge x > 0$*. Note that* `x` *is a program variable (a constant in* Σ *of sort* `Var`*), while* x *is a mathematical variable standing for integer numbers.*

If $\gamma = \langle$`skip`$, x \mapsto 10\rangle$ *and* $\rho(x) = 10$*, we have that* $(\gamma, \rho) \models \varphi$ *because* $(\gamma, \rho) \models \langle$`skip`$, x \mapsto x\rangle$ *and* $(\gamma, \rho) \models x > 0$*. If* $\rho'(x) = 11$*, we have that* $(\gamma, \rho') \not\models \varphi$*, as the matching will fail. If* $\gamma' = \langle$`skip`$, x \mapsto 0\rangle$ *and* $\rho''(x) = 0$*, we have that* $(\gamma', \rho'') \not\models \varphi$ *because, even if the match succeeds,* (γ', ρ'') *is not a model of the condition* $x > 0$*.*

To summarize the above, φ *will match exactly those* I*MP configurations that contain terminated programs (because to code part contains only the* `skip` *instruction) and in which the program variable* x *contains a strictly positive value.*

Definition 4. *A matching logic formula* φ *is called* structureless *if it does not contain any basic pattern* π*. A matching logic formula* φ *is called* pure *if* $\varphi = \pi \wedge \varphi'$*, where* π *is a basic pattern and* φ' *is a structureless matching logic formula. Note that structureless formulae are essentially FOL formulae.*

Matching logic is useful in reasoning about program configurations but it is also possible to define the operational semantics of languages in terms of matching logic formulae.

Definition 5. *A* reachability rule $\varphi \Rightarrow \varphi'$ *is a pair of two matching logic formulae* φ *and* φ'*.*

Intuitively, a reachability rule $\varphi \Rightarrow \varphi'$ states that a configuration matching φ will advance into a configuration matching φ'. Reachability rules can be used both for defining the language semantics (shown below) as well as for stating (partial) correctness properties (explained in Definition 9).

Example 3. *Take the reachability rule* $\langle$`skip;`$c, h\rangle \Rightarrow \langle c, h\rangle$ *where* c *is a (mathematical) variable of sort* `Code` *and* h *is a (mathematical) variable of sort* `Map`$\{$`Var, Int`$\}$*. The rule describes (part of) the semantics of a sequence (function symbol* `_;_`*) of* `Stmts`*. It says that when the first statement in the sequence is the empty statement and the second statement is the variable* c*, the configuration moves in one step into another configuration where only the code* c *remains to be executed in the code part and where the heap* h *remains the same.*

Another example is the rule for assignment: $\langle x$`:=`$i, h\rangle \Rightarrow \langle$`skip`$; h[i/x]\rangle$*. In this rule,* x *is a mathematical variable of sort* `Var` *(therefore it can be instantiated with program variables such as* `x`*,* `y`*, and so on),* i *is a mathematical variable of sort* `Int` *and* h *is a mathematical variable of sort* `Map`$\{$`Var, Int`$\}$*. The rule says that when the only code to execute is an assignment of a integer number* i *to a variable* x*, the heap is updated so that the value of* x *becomes* i *and the code is changed into* `skip`*. Note that the variables* x, i, h *are "shared" between the two matching logic formulae of the reachability rule, in the sense that they must be instantiated with the same values on both sides for the rule to make sense (this intuition will be formalized below).*

Definition 6. *A reachability rule* $\varphi \Rightarrow \varphi'$ *is called* pure *if both* φ *and* φ' *are pure.*

A (possibly infinite) set $\mathcal{A}$ of pure reachability rules can capture the entire operational semantics of a programming language as shown in [6]. A set $\mathcal{A}$ of reachability rules generates a transition system on $\mathcal{T}_{Cfg}$:

Definition 7. *Given a matching logic semantics domain*

$(Cfg, S, \Sigma, \mathcal{T})$ and a set $\mathcal{A}$ of reachability rules, the relation $\rightarrow_{\mathcal{A}}$ on $\mathcal{T}_{Cfg}$ is defined as follows:

- *$\gamma \rightarrow_{\mathcal{A}} \gamma'$ if there exists a rule $\varphi \Rightarrow \varphi' \in \mathcal{A}$ and a valuation ρ such that $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$ (for all $\gamma, \gamma' \in \mathcal{T}_{Cfg}$).*

Note that because the same valuation ρ is used to check satisfaction for both sides φ and φ' of the reachability rule, the free variables shared by φ and φ' must be instantiated by the same value. The relation $\rightarrow_{\mathcal{A}}$ is the one-step transition relation of program configurations and defines the dynamic behavior of the programs. We call this style of giving the semantics of a language *matching logic semantics*:

Definition 8. *If $(Cfg, S, \Sigma, \mathcal{T})$ is a matching logic semantic domain and $\mathcal{A}$ is a set of reachability rules, then the tuple $(Cfg, S, \Sigma, \mathcal{T}, \mathcal{A})$ is called a* matching logic semantics*.*

As we have already mentioned, it is known [6] that matching logic semantics can be used to describe any language. In order to achieve this, the set $\mathcal{A}$ of reachability rules can be infinite. In Figure 1 (next page), we summarize the set of reachability rules $\mathcal{A}_{\mathbf{I}_{MP}}$ of IMP. The variable x is of sort `Var`, h of sort `Map{Var, Int}`, i, i_1, i_2 of sort `Int`, e of sort `ExpI`, s, s_1, s_2 of sort `Stmt` and *code*, *code'* of sort `Code`. For the second rule, note that `op` ranges over all operands in the language. By $\texttt{op}_{Int}$ we denote the mathematical function associated to `op`. Therefore, while `_+_` is a function symbol in our language, by $\_+_{Int}\_$ we denote actual addition on integers. Therefore, the second rule will make expressions like $1 + 3$ turn into the integer $4 = 1 +_{Int} 3$. The last reachability rule is actually a schema of rules (this is why $\mathcal{A}_{\mathbf{I}_{MP}}$ is infinite). The schema makes rules in $\mathcal{A}_{\mathbf{I}_{MP}}$ closed under the set of evaluation contexts defined by C.

It is known [9], [8], [7] that, given the matching logic semantics of a language, it is possible to derive partial correctness properties of programs written in that language using a language-independent proof system called reachability logic. The reachability logic proof system is similar in purpose with Hoare logic, but it is better in the sense that it is parameterized by the matching logic semantics of the language and therefore usable for any language (for which the matching logic semantics is available). In fact, we have shown [9], [8] that the proof system is sound (it only derives sound partial correctness results) and relatively complete (any partial correctness result is derivable, relative to an oracle for doing domain reasoning).

The reachability logic proof system is summarized in Figure 2. We briefly explain how each rule works, but this is not the main purpose of this article; for a full explanation please consult our previous work [8].

The proof system derives intermediate sequents of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$ where $\mathcal{A}$ (axioms) and C (called *circularities*) are sets of reachability rules and $\varphi \Rightarrow \varphi'$ is the resulting reachability rule. The circularities are reachability rules that are assumed to hold after at least one program step takes place. The AXIOM rule allows to use the reachability rules defining

$$\textsc{Axiom}\ \frac{\varphi \Rightarrow \varphi' \in \mathcal{A} \quad \psi \text{ structureless}}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

$$\textsc{Reflexivity}\ \frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

$$\textsc{Transitivity}\ \frac{A \vdash_C \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{A} \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

$$\textsc{Conseq}\ \frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash_C \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

$$\textsc{Case Analysis}\ \frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \quad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

$$\textsc{Abstraction}\ \frac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \quad X \cap \mathit{FreeVar}(\varphi') = \emptyset}{\mathcal{A} \vdash_C (\exists X.\varphi) \Rightarrow \varphi'}$$

$$\textsc{Circularity}\ \frac{\mathcal{A} \vdash_{C \cup \varphi \Rightarrow \varphi'} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

Fig. 2. The reachability logic proof system

the operational semantics of the language in the presence of any structureless (i.e. without patterns) matching logic formula ψ. The REFLEXIVITY rule allows to take zero steps (when the set C of circularities is empty). TRANSITIVITY allows to prove $\varphi_1 \Rightarrow \varphi_2$ by first proving $\varphi_1 \Rightarrow \varphi_2$ and then $\varphi_2 \Rightarrow \varphi_3$. Note that in proving $\varphi_2 \Rightarrow \varphi_3$ the set of circularities is added to the set of axioms (as we have taken at least one step from φ_1 to φ_2, the circularities are now valid). The rule CONSEQ allows to perform domain reasoning. If φ_1' is a semantic consequence ($\models \varphi_1 \rightarrow \varphi_1'$) of φ_1 (in first-order logic) and φ_2 is a semantic consequence of φ_2', it is sufficient to show $\varphi_1' \Rightarrow \varphi_2'$ in order to obtain $\varphi_1 \Rightarrow \varphi_2$ (φ_1 is stronger than φ_1' and φ_2' is stronger than φ_2). CASE ANALYSIS allows to split the proof into two cases and ABSTRACTION allows to hide irrelevant details of φ behind an existential quantifier. Finally, CIRCULARITY is useful for proving circular behaviors such as loops or recursion. It says that in order to prove $\varphi \Rightarrow \varphi'$, we are allowed to use the same rule as a circularity. This is valid for partial correctness since circularities are used as axioms only after a strict program steps takes place (in rule TRANSITIVITY. For more details about this proof system, including its full proofs of soundness and relative completeness, please consult our previous work [8].

When C is empty, we write $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and deducing such a sequent means that $\varphi \Rightarrow \varphi'$ is a reachability rule that is a semantic consequence of the set of reachability rules $\mathcal{A}$ taken as axioms (the semantics of the programming language):

Definition 9. *We say that $\mathcal{A} \models \varphi \Rightarrow \varphi'$ (read as $\varphi \Rightarrow \varphi'$ is a semantic consequence of $\mathcal{A}$) if for any $\gamma \in \mathcal{T}$ and for*

$\langle x,\ h\rangle \Rightarrow \langle h(x),\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle i_1\ \texttt{op}\ i_2,\ h\rangle \Rightarrow \langle i_1\ \texttt{op}_{Int}\ i_2,\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$ where op ranges over the operators: +, *, %, -
$\langle x\ \texttt{:=}\ i,\ h\rangle \Rightarrow \langle \texttt{skip},\ h[i/x]\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle \texttt{skip;}s,\ h\rangle \Rightarrow \langle s,\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle \texttt{if i then}\ s_1\ \texttt{else}\ s_2,\ h\rangle \wedge i \neq 0 \Rightarrow \langle s_1,\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle \texttt{if 0 then}\ s_1\ \texttt{else}\ s_2,\ h\rangle \Rightarrow \langle s_2,\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle \texttt{while}\ e\ \texttt{do}\ s,\ h\rangle \Rightarrow \langle \texttt{if}\ e\ \texttt{then}\ s\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{else skip},\ h\rangle \in \mathcal{A}_{\mathbf{I}MP}$
$\langle C[code],\ h\rangle \Rightarrow \langle C[code'],\ h'\rangle \in \mathcal{A}_{\mathbf{I}MP}$ if $\langle code,\ h\rangle \Rightarrow \langle code',\ h'\rangle \in \mathcal{A}_{\mathbf{I}MP}$
where C ::= _ | C op e | i op C | if C then s_1 else s_2 | v := C | C;s

Fig. 1. The set of reachability rules $\mathcal{A}_{\mathbf{I}MP}$ for the IMP language.

any valuation ρ such that γ terminates in $\rightarrow_{\mathcal{A}}^*$ and such that $(\gamma, \rho) \models \varphi$ we have that $\gamma \rightarrow_{\mathcal{A}}^* \gamma'$ such that $(\gamma', \rho) \models \varphi'$.

Note that the above definition interprets reachability rules as partial correctness properties: a terminating configuration matching φ will advances into a configuration matching φ'.

The fact that the proof system is sound simply means that $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{A} \models \varphi \Rightarrow \varphi'$. The goal here is not to explain how the above proof system works or why it is sound (we have shown this in earlier papers), but merely to state that it is possible to use it in order to soundly derive partial correctness properties for any language for which the matching logics semantics is available.

For example, it is possible to show for the IMP program SUM (where SUM = while n do (s:=s+n; n:=n-1)) that, if it terminates, it indeed computes the sum of the first n natural numbers (if the starting n is not negative): $\mathcal{A}_{\mathbf{I}MP} \vdash \langle \texttt{SUM}, \texttt{n} \mapsto n, \texttt{s} \mapsto 0\rangle \wedge n \geq_{Int} 0 \Rightarrow \langle \texttt{skip}, \texttt{n} \mapsto 0, \texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2\rangle$ (the *SUM* program starting with the program variable n mapped to the integer n and the program variable s mapped to the integer 0 ends with s mapped to $n *_{Int} (n +_{Int} 1)/_{Int} 2$ and n mapped to 0).

III. Language Aggregation

In this section, we will show that, given the matching logic semantics of two languages (the left language and the right language), it is possible to construct a matching logic semantics for the *aggregated* language, in which configurations consist of pairs of configurations (C_L, C_R) of configurations C_L of the left language and configurations C_R of the right language. The main property of the aggregated semantics is that an aggregated configuration (C_L, C_R) will reach another aggregated configuration(C'_L, C'_R) (in the aggregated language) iff C_L reaches C'_L in the left language and C_R reaches C'_R in the right language.

The construction of the aggregated language is important because the equivalence of programs from the left language and respectively the right language will be reduced to partial correctness properties of programs in the aggregated language. The main difficulty in the construction of the aggregated language is making sure that elements common to both languages are mapped into the same elements in the target language. This is necessary in order to express equivalence properties. For example, if both languages have a sort for integer numbers, in the aggregated language the two sorts must be mapped into the same sort. Furthermore, the model of configurations in the aggregated language needs to agree on this sort with the two models of the initial languages.

We next describe how to construct the aggregated language. Part of this construction is given, with full proofs, in [10], [11], where it is used for the purpose of establishing *mutual equivalence*, a type of equivalence that also takes into account termination.

We will assume that we are given two matching logic semantics $(Cfg_L, S_L, \Sigma_L, \mathcal{T}_L, \mathcal{A}_L)$ and $(Cfg_R, S_R, \Sigma_R, \mathcal{T}_R, \mathcal{A}_R)$ for two languages that agree on a common part: there exists a set of sorts S_0, a signature Σ_0, a model T_0 and two morphisms $h_L : (S_0, \Sigma_0) \rightarrow (S_L, \Sigma_L)$ and $h_R : (S_0, \Sigma_0) \rightarrow (S_R, \Sigma_R)$. The common sorts S_0 typically contain the shared sorts between the two languages (such as integers, floats, strings, etc.) and Σ_0 contains the common function symbols (such as addition of integers). The morphisms h_R and h_L simply rename the sorts (resp. function symbols) from S_0 (respectively Σ_0) such that they match the sorts/function symbols in S_L and S_R (resp. Σ_L and Σ_R).

We will construct the matching logic semantics $(Cfg, S, \Sigma, \mathcal{T}, \mathcal{A})$ of the aggregated language in several steps:

1) we will show how to choose S and Σ (by using the pushout theorem),
2) we will show how to construct the aggregated model $\mathcal{T}$ using the FOL amalgamation theorem,
3) we will show how to extend S, Σ and $\mathcal{T}$ in order to account for the sort *Cfg* of aggregated configurations and
4) finally we will show how to construct $\mathcal{A}$ from $\mathcal{A}_L$ and $\mathcal{A}_R$.

a) Step 1.: As we mentioned, firstly we will show how to construct S and Σ. The construction of S and Σ is given by the push-out theorem:

Theorem 1 (pushout)**.** *Let (S_R, Σ_R), (S_L, Σ_L) and (S_0, Σ_0) be three FOL signatures, h_R a morphism from (S_0, Σ_0) to (S_R, Σ_R) and h_L a morphism from (S_0, Σ_0) to (S_L, Σ_L). There exists a tuple $(h'_L, (S', \Sigma'), h'_R)$, called the* pushout

of $(S_L, \Sigma_L) \xleftarrow{h_L} (S_0, \Sigma_0) \xrightarrow{h_R} (S_R, \Sigma_R)$*, where* h'_L *is a morphism from* (S_L, Σ_L) *to* (S', Σ') *and* h'_R *a morphism from* (S_R, Σ_R) *to* (S', Σ') *such that the following conditions hold:*

1) *(commutativity)* $h'_L(h_L(x)) = h'_R(h_R(x))$ *for all* $x \in S_0 \cup \Sigma_0$ *and*
2) *(minimality) if there exist* S'', Σ'' *and morphisms* h''_L *(from* (S_L, Σ_L) *to* (S'', Σ'')*) and* h''_R *(from* (S_R, Σ_R) *to* (S'', Σ'')*) such that* $h''_L(h_L(x)) = h''_R(h_R(x))$ *for all* $x \in S_0 \cup \Sigma_0$ *then there exists a morphism* h *from* (S', Σ') *to* (S'', Σ'').

Furthermore, the pushout is unique (up to renaming).

The proof can be found in [10].

We will choose $S = S' \uplus \{Cfg\}$ to contain the sorts S' provided by push-out theorem and a fresh sort *Cfg* for configurations. The signature Σ is chosen to include Σ': $\Sigma = \Sigma' \uplus \Sigma_{Cfg}$, where Σ_{Cfg} will be defined later.

b) Step 2.: The next step is to choose the right model $\mathcal{T}$. The main difficulty is making sure that the sorts that are shared between the two languages are given the same interpretation in $\mathcal{T}$. Fortunately, the existance of this model $\mathcal{T}$ is given by the *FOL amalgamation theorem*:

Theorem 2 (amalgamation)**.** *If* $\mathcal{T}_R$*,* $\mathcal{T}_L$ *and* $\mathcal{T}_0$ *are models of* (S_R, Σ_R)*,* (S_L, Σ_L) *and respectively* (S_0, Σ_0) *such that* $\mathcal{T}_R\restriction_{h_R} = \mathcal{T}_L\restriction_{h_L} = \mathcal{T}_0$*, there exists a unique model* $\mathcal{T}'$ *of* (S', Σ') *such that* $\mathcal{T}'\restriction_{h'_L} = \mathcal{T}_L$ *and* $\mathcal{T}'\restriction_{h'_R} = \mathcal{T}_R$.

By $\mathcal{T}_R\restriction_{h_r}$ we denote the reduct of $\mathcal{T}_R$ through the morphism h_R. The proof can be found in [10].

The amalgamation theorem gives us a model $\mathcal{T}'$ that agrees with $\mathcal{T}_L$ and $\mathcal{T}_R$ on the shared parts. We will extend $\mathcal{T}'$ into $\mathcal{T}$ as described in the next step.

c) Step 3.: In this step we show how to account for the aggregated sort *Cfg*.

We will start from $S', \Sigma', \mathcal{T}'$ defined above and build S, Σ and $\mathcal{T}$.

Firstly, $S = S' \uplus \{Cfg\}$. For the set of sorts, we simply use the set of sorts S' given to us by the pushout theorem (this sort will contain all sorts from S_L and S_R (possibly renamed by the morphism) such that common sorts (in S_0) have the same name), to which we add the new sort *Cfg* of configurations in the aggregated language.

Secondly, $\Sigma = \Sigma' \uplus \{(\_,\_), pr_L(\_), pr_R(\_)\}$. The distinguished binary function symbol $(\_,\_) : h'_L(Cfg_L) \times h'_R(Cfg_R) \to Cfg$ constructs aggregated configurations from the left and the right configuration, while the distinguished unary projection symbols $pr_L : Cfg \to h'_L(Cfg_L)$ and $pr_R : Cfg \to h'_R(Cfg_R)$ return the left-hand side and respectively the right-hand side of the aggregated configuration.

Thirdly, we define the model $\mathcal{T}$ of (S, Σ). We start from the model $\mathcal{T}'$ of (S', Σ') given by the amalgamation theorem in **Step 2** and we provide the interpretation of the new sorts/symbols:

1) $T_{Cfg} = T'_{h'_L(Cfg_L)} \times T'_{h'_R(Cfg_R)}$ (the set of configurations is the cartesian product of the sets of left configurations and right configurations),
2) $T_{(\_,\_)}(\gamma_L, \gamma_R) = (\gamma_L, \gamma_R)$ (the interpretation of the pairing function symbol applied to γ_L and γ_R is to simply construct the pair of the two elements),
3) $T_{pr_L(\_)}((\gamma_L, \gamma_R)) = \gamma_L$ (the interpretation of the left projection applied over the configuration (γ_L, γ_R) is simply γ_L),
4) $T_{pr_R(\_)}((\gamma_L, \gamma_R)) = \gamma_R$ (the interpretation of the right projection applied over the configuration (γ_L, γ_R) is simply γ_R) and
5) $T_x = T'_x$ for any other object $x \in S' \cup \Sigma'$.

The above construction satisfies a very important property, which we will use in order to prove that the aggregated language behaves properly.

Proposition 1. *Let* $(\gamma_L, \gamma_R) \in \mathcal{T}_{Cfg}$ *be a configuration. Let* φ *be a pure matching logic formula with no variables of sort Cfg. For any valuation* $\rho : Var \to \mathcal{T}$*, we have that* $((\gamma_L, \gamma_R), \rho) \models \varphi$ *iff* $(\gamma_L, \rho) \models pr_L(\varphi)$ *and* $(\gamma_R, \rho) \models pr_R(\varphi)$.

See [10] for the proof.

d) Step 4.: Next we show how to construct $\mathcal{A}$ from $\mathcal{A}_L$ and $\mathcal{A}_R$. This step is new when compared to [10], [11], which only contains a way to aggregate the *configurations* of the two languages. This step allows to aggregate the actual operational semantics, in addition to what was known before.

We recall that $\mathcal{A}_L$ and $\mathcal{A}_R$ contain pure reachability logic formulae (i.e. contain only matching logic formulae φ of the form $\varphi = \pi \wedge \varphi'$ where π is a basic pattern and φ' is structureless)). We define $\mathcal{A}$ to be the set:

$$\begin{aligned}\mathcal{A} \quad = \quad & \{(h'_L(\pi_L), x_R) \wedge \varphi \Rightarrow (h'_L(\pi'_L), x_R) \wedge \varphi' \mid \\ & \quad \pi_L \wedge \varphi \Rightarrow \pi'_L \wedge \varphi' \in \mathcal{A}_L\} \ \cup \\ & \{(x_L, h'_R(\pi_R)) \wedge \varphi \Rightarrow (x_L, h'_R(\pi'_R)) \wedge \varphi' \mid \\ & \quad \pi_R \wedge \varphi \Rightarrow \pi'_R \wedge \varphi' \in \mathcal{A}_R\} \ \cup\end{aligned}$$

In the above definition, x_L and respectively x_R are variables of sort $h'_L(Cfg_L)$ and respectively $h'_R(Cfg_R)$. The main property of the set of reachability rules $\mathcal{A}$ is:

Lemma 1. *We have that* $(\gamma_L, \gamma_R) \to_{\mathcal{A}} (\gamma'_L, \gamma'_R)$ *iff either* $\gamma_L \to_{\mathcal{A}_L} \gamma'_L$ *and* $\gamma_R = \gamma'_R$ *or* $\gamma_R \to_{\mathcal{A}_R} \gamma'_R$ *and* $\gamma_L = \gamma'_L$ *(exclusively).*

Proof. We prove each implication separately.

e) Direct implication.: For the direct implication, assume $(\gamma_L, \gamma_R) \to_{\mathcal{A}} (\gamma'_L, \gamma'_R)$. By the definition of $\to_{\mathcal{A}}$, there must exist a rule $\pi \wedge \varphi \Rightarrow \pi' \wedge \varphi'$ and a valuation ρ such that $((\gamma_L, \gamma_R), \rho) \models \pi \wedge \varphi$ and $((\gamma'_L, \gamma'_R), \rho) \models \pi' \wedge \varphi'$. By the definition of $\mathcal{A}$, we have that either $\pi = (h'_L(\pi_L), x_R)$ and $\pi' = (h'_L(\pi'_L), x_R)$ for a variable x_R and for a rule $\pi_L \wedge \varphi \Rightarrow \pi'_L \wedge \varphi' \in \mathcal{A}_L$ or $\pi = (x_L, h'_R(\pi_R))$ and $\pi' = (x_L, h'_R(\pi'_R))$ for a variable x_L and for a rule $\pi_R \wedge \varphi \Rightarrow \pi'_R \wedge \varphi' \in \mathcal{A}_R$. We will only discuss the first case, as the second is completely analagous. Assume therefore that $\pi = (h'_L(\pi_L), x_R)$ and $\pi' = (h'_L(\pi'_L), x_R)$ for a variable x_R and for a rule $\pi_L \wedge \varphi \Rightarrow \pi'_L \wedge \varphi' \in \mathcal{A}_L$. As $((\gamma_L, \gamma_R), \rho) \models \pi \wedge \varphi$ and $((\gamma'_L, \gamma'_R), \rho) \models \pi' \wedge \varphi'$, we have that $((\gamma_L, \gamma_R), \rho) \models (h'_L(\pi_L), x_R)) \wedge \varphi$ and $((\gamma'_L, \gamma'_R), \rho) \models (h'_L(\pi'_L), x_R)) \wedge \varphi'$. We have therefore that

$(\gamma_L, \rho) \models \pi_L \wedge \varphi$, $(\gamma'_L, \rho) \models \pi'_L \wedge \varphi'$ and $\gamma_R = x_R\rho = \gamma'_R$ (which is what we had to show).

f) Reverse implication.: For the reverse implication, assume that $\gamma_L \rightarrow_{\mathcal{A}_L} \gamma'_L$ and $\gamma_R = \gamma'_R$ (the other case is analagous). We have that there exist a valuation ρ and a rule $\pi_L \wedge \varphi \Rightarrow \pi'_L \wedge \varphi' \in \mathcal{A}_L$ such that $(\gamma_L, \rho) \models \pi_L \wedge \varphi$ and $(\gamma'_L, \rho) \models \pi'_L \wedge \varphi'$. We then have $((\gamma_L, \gamma_R), \rho[\gamma_R/x_R]) \models (h'_L(\pi_L), x_R) \wedge \varphi$ and $((\gamma'_L, \gamma_R), \rho[\gamma_R/x_R]) \models (h'_L(\pi'_L), x_R) \wedge \varphi'$. But by definition, $(h'_L(\pi_L), x_R) \wedge \varphi \Rightarrow (h'_L(\pi'_L), x_R) \wedge \varphi' \in \mathcal{A}$ and therefore $(\gamma_L, \gamma_R) \rightarrow_{\mathcal{A}} (\gamma'_L, \gamma_R)$, which is what we had to show (as $\gamma_R = \gamma'_R$).

□

Using the above lemma as a helper, it easily follows that:

Theorem 3 (aggregation)**.** *We have that* $(\gamma_L, \gamma_R) \rightarrow^*_{\mathcal{A}} (\gamma'_L, \gamma'_R)$ *iff* $\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$ *and* $\gamma_L \rightarrow^*_{\mathcal{A}_R} \gamma'_R$.

Proof. We prove each implication separately.

g) Direct implication.: For the direct implication, assume that $(\gamma_L, \gamma_R) \rightarrow^*_{\mathcal{A}} (\gamma'_L, \gamma'_R)$. There exist $n \geq 0, (\gamma^1_L, \gamma^1_R), \ldots, (\gamma^n_L, \gamma^n_R) \in \mathcal{T}_{Cfg}$ such that $(\gamma_L, \gamma_R) = (\gamma^1_L, \gamma^1_R) \rightarrow_{\mathcal{A}} (\gamma^2_L, \gamma^2_R) \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} (\gamma^n_L, \gamma^N_R) = (\gamma'_L, \gamma'_R)$. By Lemma 1, we have that, for all $1 \leq i < n$, either $\gamma^i_L \rightarrow_{\mathcal{A}_L} \gamma^{i+1}_L$ and $\gamma^i_R = \gamma^{i+1}_R$ or $\gamma^i_R \rightarrow_{\mathcal{A}_R} \gamma^{i+1}_R$ and $\gamma^i_L = \gamma^{i+1}_L$. Therefore, we have for all $1 \leq i < n$ that $\gamma^i_L = \gamma^{i+1}_L$ or $\gamma^i_L \rightarrow_{\mathcal{A}_L} \gamma^{i+1}_L$, which means that $\gamma_L = \gamma^1_L \rightarrow^*_{\mathcal{A}_L} \gamma^n_L = \gamma'_L$. Also we have that for all $1 \leq i < n$ that $\gamma^i_R = \gamma^{i+1}_R$ or $\gamma^i_R \rightarrow_{\mathcal{A}_R} \gamma^{i+1}_R$, which means that $\gamma_R = \gamma^1_R \rightarrow^*_{\mathcal{A}_R} \gamma^n_R = \gamma'_R$. In conclusion, we have that $\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$ and that $\gamma_R \rightarrow^*_{\mathcal{A}_R} \gamma'_R$, which is what we had to show.

h) Reverse implication.: For the reverse implication, assume that $\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$ and $\gamma_L \rightarrow^*_{\mathcal{A}_R} \gamma'_R$. From$\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$, we have that there exist $n \geq 0, \gamma^1_L, \ldots, \gamma^n_L$ such that $\gamma_L = \gamma^1_L \rightarrow_{\mathcal{A}_L} \gamma^2_L \rightarrow_{\mathcal{A}_L} \ldots \rightarrow_{\mathcal{A}_L} \gamma^n_L = \gamma'_L$. By Lemma 1, we have that $(\gamma_L, \gamma_R) = (\gamma^1_L, \gamma_R) \rightarrow_{\mathcal{A}} (\gamma^2_L, \gamma_R) \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} (\gamma^n_L, \gamma_R) = (\gamma'_L, \gamma_R)$ and therefore $(\gamma_L, \gamma_R) \rightarrow^*_{\mathcal{A}} (\gamma'_L, \gamma_R)$.

From$\gamma_R \rightarrow^*_{\mathcal{A}_R} \gamma'_R$, we have that there exist $m \geq 0, \gamma^1_R, \ldots, \gamma^m_R$ such that $\gamma_R = \gamma^1_R \rightarrow_{\mathcal{A}_R} \gamma^2_R \rightarrow_{\mathcal{A}_R} \ldots \rightarrow_{\mathcal{A}_R} \gamma^m_R = \gamma'_R$. By Lemma 1, we have that $(\gamma'_L, \gamma_R) = (\gamma'_L, \gamma^1_R) \rightarrow_{\mathcal{A}} (\gamma'_L, \gamma^2_R) \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} (\gamma'_L, \gamma^m_R) = (\gamma'_L, \gamma'_R)$ and therefore $(\gamma'_L, \gamma_R) \rightarrow^*_{\mathcal{A}_L} (\gamma'_L, \gamma'_R)$.

From $(\gamma_L, \gamma_R) \rightarrow^*_{\mathcal{A}} (\gamma'_L, \gamma_R)$ and $(\gamma'_L, \gamma_R) \rightarrow^*_{\mathcal{A}_L} (\gamma'_L, \gamma'_R)$, we obtain by transitivity our conclusion: $(\gamma_L, \gamma_R) \rightarrow^*_{\mathcal{A}} (\gamma'_L, \gamma'_R)$.

□

In this section, we have shown how to construct a matching logic semantics $(Cfg, S, \Sigma, \mathcal{T}, \mathcal{A})$ of the aggregated language from the matching logic semantics $(Cfg_L, S_L, \Sigma_L, \mathcal{T}_L, \mathcal{A}_L)$ and $(Cfg_R, S_R, \Sigma_R, \mathcal{T}_R, \mathcal{A}_R)$ of the left language and respectively the right language, assuming that the left and the right languages agree on a common part $(S_0, \Sigma_0, \mathcal{T}_0)$.

We reuse previous work [10], [11] in order to aggregate the static part of the languages (the configurations) and we show how to aggregate the sets of reachability rules $\mathcal{A}_L$ and $\mathcal{A}_R$ in order for the aggregated language to behave as expected. This property of the aggregated language is captured in Theorem 3, which states that an aggregated configuration transitions into another aggregated configuration if and only if the individual configurations take similar transitions in the starting languages and which will be critical to show how to reduce partial equivalence to partial correctness in the next section.

IV. Reducing Partial Equivalence to Partial Correctness

In this section, we will show how to specify partial equivalence in terms of matching logic and how partial equivalence of a program written in the left language and a program written in the right language can be reduced to partial correctness of a program in the aggregated language.

i) Specifying partial equivalence.: Two programs P_L and P_R are partially equivalent if, when given the same input, they produce the same output if they both terminate.

Therefore, in order to specify partial equivalence, we need a way to provide input to and extract output from the programs. We will consider two sets I and O of input value and respectively output values. We do not make any assumption on I and O; these sets are chosen depending on what is the exact input and output of the program being considered.

We will also consider two functions $input_L : I \rightarrow \mathcal{T}_{h'_L(Cfg_L)}$ and $input_R : I \rightarrow \mathcal{T}_{h'_R(Cfg_R)}$ that, given the value of the input, produce the initial configurations of the two programs P_L and P_R.

Example 4. *Let us consider that the left language is* IMP *and that the left program is the program* `SUM = while n do (s:=s+n; n:=n-1)` *defined previously.*

We will consider that the right language is also IMP *and that the right program is a program* `SUM'` *that also computes the sum of the first* n *natural numbers, but in a different way:* `SUM' = i:=0;while i<=n do(t:=t+i; i:=i+1)`.

In this case, we want the input to our programs to be the value of the program variable `n`*. We will consider therefore that the set* $I = \mathbb{N}$ *is the set of all naturals. The input function for the left language is* $input_L(n) = \langle$`SUM`$, \texttt{n} \mapsto n\rangle$ *and the input function for the right language is* $\texttt{input}_R(n) = \langle$`SUM'`$, \texttt{n} \mapsto n\rangle$*. In this case, the input functions simply place the programs* `SUM` *and* `SUM'` *in the correct place in the initial configurations and initialize in the heap the value of the program variables* `n` *with the value of the input.*

But we also need a way to compare generically the output of the programs. Therefore we need a language-indepedent way of extracting the result from a terminated configuration. We will assume that there exist partial functions $output_L : \mathcal{T}_{h'_L(Cfg_L)} \rightarrow O$ and $output_R : \mathcal{T}_{h'_R(Cfg_R)} \rightarrow O$ that take as input a terminated configuration and return the result as a member of the set O of outputs. We do not make any assumption on these functions and they are provided as a parameter by the user. These functions are partial because they only return the result for terminated configurations (in which the result is available).

Example 5. *Continuing the previous example, the result of* `SUM` *and* `SUM'` *is an integer. Therefore we choose* $O = \mathbb{Z}$ *and*

we will use the partial function $output_L(\langle skip, s \mapsto s, h\rangle) = s$ *to extract the result of the computation in the case of* SUM *and* $output_R(\langle skip, t \mapsto t, h\rangle) = t$ *in the case of* SUM′*. Note that* h *is a variable matching the rest of the heap.*

Now that we have the input and output function available, we can define what it means for two programs to be partially equivalent:

Definition 10 (partial equivalence)**.** *Given the input and output functions* $input_L, input_R, output_L, output_L$ *of the left program* P_L *and of the right program* P_R*, we say that* P_L *and* P_R *are partially equivalent, if, for all inputs* $i \in I$ *such that* $input_L(i)$ *and* $input_R(i)$ *terminate, we have that* $input_L(i) \rightarrow^*_{\mathcal{A}_L} c_L$ *and* $input_R(i) \rightarrow^*_{\mathcal{A}_R} c_R$ *such that* $output_L(c_L) = output_R(c_R)$*.*

That is, two programs are partially equivalent, if, when both terminate from their initial configurations in which they have the same input if they reach final configurations c_L, c_R in which their output is the same.

j) Reducing partial equivalence to partial correctness.: We will now show how to prove partial equivalence of two programs using partial correctness in the aggregated language.

Let $input_L, input_R, output_L, output_L$ be the input/output functions of the left program P_L and respectively of the right program P_R.

Let i be a mathematical variable (of sort I) and let $\varphi = (input_L(i), input_R(i))$. Note that φ is a matching logic formula over the aggregated language.

Let c_L and c_R be terminated configuration (on which $output_L$ and respectively $output_R$ is defined). Let φ' be the matching logic formula $(c_L, c_R) \wedge output_L(c_L) = output_R(c_R)$. Note that φ' is also a matching logic formula over the aggregated language.

Theorem 4 (reduction)**.** *The programs* P_L *and* P_R *are partially equivalent iff* $\mathcal{A} \models \varphi \Rightarrow \varphi'$*.*

Proof. Assume that P_L and P_R are partially equivalent. Then, by definition, for all inputs $i \in I$ such that $input_L(i)$ and $input_R(i)$ terminate, we have that $input_L(i) \rightarrow^*_{\mathcal{A}_L} c_L$ and $input_R(i) \rightarrow^*_{\mathcal{A}_R} c_R$ such that $output_L(c_L) = output_R(c_R)$. Let i be an arbitrary input such that $input_L(i)$ and $input_R(i)$ terminate. We have that $input_L(i) \rightarrow_{\mathcal{A}_L} c_L$ and $input_R(i) \rightarrow_{\mathcal{A}_R} c_R$ such that $output_L(c_L) = output_R(c_R)$ and, by Theorem 3, $(input_L(i), input_R(i))$ terminates and furthermore $(input_L(i), input_R(i)) \rightarrow^*_{\mathcal{A}} (c_L, c_R)$. But $output_L(c_L) = output_R(c_R)$ and therefore $\mathcal{A} \models (input_L(i), input_R(i)) \Rightarrow (c_L, c_R) \wedge output(c_L) = output(c_R)$. Vice-versa, assume that $\mathcal{A} \models \varphi \Rightarrow \varphi'$. Let $(input_L(i), input_R(i))$ be a terminating configuration and ρ a valution such that $((input_L(i), input_R(i)), \rho) \models \varphi$. As $\mathcal{A} \models \varphi \Rightarrow \varphi'$, it follows that there exists a configuration (c_L, c_R) such that $(input_L(i), input_R(i)) \rightarrow^*_{\mathcal{A}} (c_L, c_R)$ and $((c_L, c_R), \rho) \models \varphi'$. Therefore $output_L(c_L) = output_R(c_R)$. By Theorem 3, we obtain that $input_L(i) \rightarrow^*_{\mathcal{A}_L} c_L$ and $input_R(i) \rightarrow^*_{\mathcal{A}_R} c_R$. Together with $output_L(c_L) = output_R(c_R)$, we have that the conditions for partialy equivalence in Definition 10 are satisfied and therefore P_L and P_R are partially equivalent. □

We conclude that in order to prove partial equivalence of the two programs, it is sufficient to derive the partial correctness property $\varphi \Rightarrow \varphi'$ in the aggregated language.

Example 6. *For the* SUM *and* SUM′ *programs, we have that* $\varphi = (\langle SUM, n \mapsto i\rangle, \langle SUM', n \mapsto i\rangle)$ *and that* $\varphi' = (\langle skip, s \mapsto s, h\rangle, \langle skip, t \mapsto t, h\rangle) \wedge s = t$*.*

Using the reachability logic proof system described in the preliminaries, we can derive that $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$*. By the soundness of the proof system, we have that* $\mathcal{A} \models \varphi \Rightarrow \varphi'$ *and, by Theorem 4, we have that* SUM *and* SUM′ *are partially equivalent.*

V. Conclusion, Related and Future Work

We have shown that the problem of partial program equivalence reduces to the problem of partial correctness in an aggregated language. We have described how to build this aggregated language constructively and how to use partial correctness tools for the aggregated language in order to derive partial equivalence properties.

It was first remarked by Hoare in [1] that program equivalence might be easier than program correctness. Among the recent works on equivalence we mention [12], [13], [14]. The first one targets programs that include recursive procedures, the second one exploits similarities between single-threaded programs in order to prove their equivalence, and the third one extends the equivalence-verification to multi-threaded programs. They use operational semantics (of a specific language they designed, called LPL) and proof systems, and formally prove their proof system's soundness. In [12] a classification of equivalence relations used in program-equivalence research is given. In [15], an implementation of a parametrized equivalence prover is presented. A way to construct the aggregation of an imperative language with itself in order to prove relational properties is given in [16].

A lot of work on program equivalence arise from the verification of compilers. One approach is full compiler verification (e.g. CompCert [17]), which is incomparable to our work since it produces computer-checked proofs of equivalence for a particular language, while our own work produces proofs (which are not computer-checked) of equivalence for any languages. Another approach is the individual verification of each particular compilation run [18]. Other work target specific classes of languages: functional [19], microcode [20], CLP [21]. In order to be less language-specific some approaches advocate the use of intermediate languages, such as [22], which works with the Boogie intermediate language. However, our approach is better, since we show how to obtain a proof system for partial correctness in the aggregated language directly from the language semantics of the original languages; therefore there is no need to trust the compiler from the original language to the target language. And finally, only a few approaches, among which [17], [20], deal with real-life language and industrial-size programs in those languages. This is in contrast to the

equivalence checking of hardware circuits, which has entered the mainstream industrial practice (see, e.g., [23] for a survey on this topic). In [24], [10], [11], proof systems for other equivalence relations between programs are given.

In constrast with all the above mentioned papers,we do not give here a proof system for partial equivalence, but we show a more powerful result: that in order to show partial equivalence, one must simply show partial correctness in a more complex language.

As future work, we intend to implement our work in order to test the practicality of the reduction and investigate whether other equivalences such as mutual equivalence or total equivalence can also be reduced to correctness properties in more complex languages.

REFERENCES

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[2] G. Roşu, C. Ellison, and W. Schulte, "Matching logic: An alternative to Hoare/Floyd logic," in *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, ser. Lecture Notes in Computer Science, M. Johnson and D. Pavlovic, Eds., vol. 6486, 2010, pp. 142–162.

[3] G. Roşu and A. Ştefănescu, "Matching logic: A new program verification approach," in *Proceedings of the 2010 Workshop on Usable Verification (UV'10)*. Microsoft Research, 2010.

[4] G. Roşu and A. Ştefănescu, "Matching Logic: A New Program Verification Approach (NIER Track)," in *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*. ACM, 2011, pp. 868–871.

[5] ——, "Towards a unified theory of operational and axiomatic semantics," in *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, ser. Lecture Notes in Computer Science, vol. 7392. Springer, 2012, pp. 351–363.

[6] ——, "From Hoare logic to matching logic reachability," in *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, ser. Lecture Notes in Computer Science, vol. 7436. Springer, 2012, pp. 387–402.

[7] ——, "Checking reachability using matching logic," in *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012, pp. 555–574.

[8] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, June 2013, pp. 358–367.

[9] A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu, "All-path reachability logic," in *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, ser. LNCS, vol. 8560. Springer, July 2014, pp. 425–440.

[10] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A language independent proof system for mutual program equivalence (revisited)," Faculty of Computer Science, Tech. Rep. TR14-01, May 2014.

[11] ——, "A language independent proof system for mutual program equivalence," in *16th International Conference on Formal Engineering Methods (ICFEM'14)*. Luxembourg: Springer, Nov. 2014.

[12] B. Godlin and O. Strichman, "Inference rules for proving the equivalence of recursive procedures," *Acta Informatica*, vol. 45, no. 6, pp. 403–439, 2008.

[13] ——, "Regression verification: proving the equivalence of similar programs," *Software Testing, Verification and Reliability*, 2012.

[14] S. Chaki, A. Gurfinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *VMCAI, LNCS 7148*, 2012, pp. 119–135.

[15] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," in *PLDI*. ACM, 2009, pp. 327–337.

[16] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, 2011, pp. 200–214.

[17] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[18] G. Necula, "Translation validation for an optimizing compiler," in *PLDI*. ACM, 2000, pp. 83–94.

[19] A. Pitts, "Operational semantics and program equivalence," in *Applied Semantics Summer School, LNCS 2395*, 2002, pp. 378–412. [Online]. Available: http://dl.acm.org/citation.cfm?id=647424.725796

[20] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck, "Formal verification of backward compatibility of microcode," in *CAV, LNCS 3576*, 2005, pp. 185–198.

[21] S. Crăciunescu, "Proving the equivalence of CLP programs," in *ICLP, LNCS 2401*, 2002, pp. 287–301.

[22] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *CAV, LNCS 7358*, 2012, pp. 712–717.

[23] F. Somenzi and A. Kuehlmann, *Electronic Design Automation For Integrated Circuits Handbook*, 2006, vol. 2, ch. 4: Equivalence Checking.

[24] D. Lucanu and V. Rusu, "Program equivalence by circular reasoning," in *IFM*, ser. Lecture Notes in Computer Science. Springer, 2013, pp. 362–377.