Programming Language Aggregation with Applications in Equivalence Checking

Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, Grigore Rosu

Programming Language Aggregation with Applications in Equivalence Checking

Ştefan Ciobâcă[1], Dorel Lucanu[1], Vlad Rusu[2], and Grigore Roşu[1,3]
[1] "Alexandru Ioan Cuza" University, Romania
[2] Inria Lille, France
[3] University of Illinois at Urbana-Champaign, USA

Abstract. We show that, given the operational semantics of two programming languages L and R, it is possible to construct an aggregate language, in which programs consist of pairs of programs from L and respectively R. In the aggregate language, a program $P = (P_L, P_R)$ takes a step from either P_L or P_R. The main difficulty is how to aggregate the two languages so that data such as integers or booleans that are common to both languages has the same representation in the aggregate language. The aggregation is based on the pushout theorem, which allows us to construct a model of the aggregate language from models of the initial languages, while making sure that the interpretation of common data such as integers is the same.
A main application of the aggregation result is in equivalence checking. It is possible to check for example that two programs P_L and P_R (written in L and respectively R) compute the same result by checking the partial correctness of the pair (P_L, P_R) in the aggregate language.

1 Introduction and Preliminaries

This abstract explains how to construct a single language that is capable of executing pairs of programs written in the two languages. The challenge is how to achieve that generically, where the two languages are given by their formal semantics, without relying on the specifics of any particular language. The aggregated language must be capable of independently "executing" pairs of programs in the original languages.

We first introduce our formalism for definining programming languages in terms of operations semantics. Our formalism is based on **matching logic**, Matching logic formulae extend FOL formulae with terms of sort *Cfg* as atomic formulae called **basic patterns**:

Definition 1. ***Matching logic formulae*** *(ML formulae) extend first-order formulae with terms π:*

$$\varphi ::= \neg\varphi, \varphi \wedge \varphi, \exists x.\varphi, \pi$$

where $\pi \in T_{Cfg}(Var)$ and $x \in Var$ is a variable.

Given a first-order model $\mathcal{T}$, matching logic formulae are interpreted in the presence of a valuation $\rho : Var \to \mathcal{T}$ as in first-order logic, but also of a element $\gamma \in \mathcal{T}$ that will be matched by terms π. The semantics of the usual logical connectives works the same way as in first-order logic, with the semantics of terms π being defined by **matching**.

Definition 2. *The* ***matching logic satisfaction*** *relation $\models$ (written as $(\gamma, \rho) \models \varphi$ and read as (γ, ρ) is a* ***model*** *of φ) is defined inductively as follows (missing cases are the same as in FOL):*

1. *$(\gamma, \rho) \models \neg\varphi'$ if it is not true that $(\gamma, \rho) \models \varphi'$*
2. *$(\gamma, \rho) \models \pi$, where π is a basic pattern if $\rho(\pi) = e$.*

For example, the ML-formulae $\langle \texttt{s := 10}, \texttt{s} \mapsto s \rangle \wedge s \geq 0$ will be satisfied by program configurations γ in which the only code to execute is an assignment (`s := 10`) and in which the heap maps the program variables `s` to an integer s that is greater than 0. Note that $\langle \cdot, \cdot \rangle$ is a function

$\langle i_1 \texttt{op}\, i_2 \rangle \Rightarrow \langle i_1 \texttt{op}_{Int} i_2 \rangle \in \mathcal{A}_F$
$\langle \texttt{if i then}\ e_1\ \texttt{else}\ e_2 \rangle \wedge i \neq 0 \Rightarrow \langle e_1 \rangle \in \mathcal{A}_F$
$\langle \texttt{if 0 then}\ e_1\ \texttt{else}\ e_2 \rangle \Rightarrow \langle e_2 \rangle \in \mathcal{A}_F$
$\langle \texttt{letrec}\ f\ x = e\ \texttt{in}\ e' \rangle \Rightarrow \langle e'[f \mapsto (\mu f.\lambda x.e)] \rangle \in \mathcal{A}_F$
$\langle (\lambda x.e)\ v \rangle \Rightarrow \langle e[x \mapsto v] \rangle \in \mathcal{A}_F$
$\langle \mu x.e \rangle \Rightarrow \langle e[x \mapsto (\mu x.e)] \rangle \in \mathcal{A}_F$
$\langle C[c] \rangle \Rightarrow \langle C[c'] \rangle \in \mathcal{A}_F$ if $\langle c \rangle \Rightarrow \langle c' \rangle \in \mathcal{A}_F$

where $C\ ::= \_ \mid C\ \texttt{op}\ e \mid \texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid C\ e \mid v\ C$

Fig. 1. Matching logic semantics of `FUN` as a set $\mathcal{A}_F$ of reachability rules schemata. *op* ranges over the binary function symbols and op_{Int} is their denotation in $\mathcal{T}_F$

$\langle x,\ env \rangle \Rightarrow \langle env(x),\ env \rangle \in \mathcal{A}_I$
$\langle i_1\ \texttt{op}\ i_2,\ env \rangle \Rightarrow \langle i_1\ \texttt{op}_{Int}\ i_2,\ env \rangle \in \mathcal{A}_I$
$\langle x\ \texttt{:=}\ i,\ env \rangle \Rightarrow \langle \texttt{skip},\ env[x \mapsto i] \rangle \in \mathcal{A}_I$
$\langle \texttt{skip;}s,\ env \rangle \Rightarrow \langle s,\ env \rangle \in \mathcal{A}_I$
$\langle \texttt{if i then}\ s_1\ \texttt{else}\ s_2,\ env \rangle \wedge i \neq 0 \Rightarrow \langle s_1,\ env \rangle \in \mathcal{A}_I$
$\langle \texttt{if 0 then}\ s_1\ \texttt{else}\ s_2,\ env \rangle \Rightarrow \langle s_2,\ env \rangle \in \mathcal{A}_I$
$\langle \texttt{while}\ e\ \texttt{do}\ s,\ env \rangle \Rightarrow \langle \texttt{if}\ e\ \texttt{then}\ s\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{else skip},\ env \rangle \in \mathcal{A}_I$
$\langle C[code],\ env \rangle \Rightarrow \langle C[code'],\ env' \rangle \in \mathcal{A}_I$ if $\langle code,\ env \rangle \Rightarrow \langle code',\ env' \rangle \in \mathcal{A}_I$

where $C\ ::= \_, C\ \texttt{op}\ e, i\ \texttt{op}\ C, \texttt{if}\ C\ \texttt{then}\ s_1\ \texttt{else}\ s_2, v\ \texttt{:=}\ C, C\ s$

Fig. 2. Matching logic semantics of `IMP` as a set $\mathcal{A}_I$ of reachability rules (schemata). *op* ranges over the binary function symbols and op_{Int} is their denotation in $\mathcal{T}_I$.

symbol for constructing configurations, the first parameter being the code to execute and the second parameter being a map representing the heap.

It has been shown (see, e.g., [4]) that ordered pairs of ML-formulae can be used to define any programming language semantics. For example, the pair:

$$\langle var := val, M \rangle \Rightarrow \langle \texttt{skip}, M[val/var] \rangle$$

defines the semantics of assignment: whenever an assignment is to be executed, the assignment instruction is consumed (hence the `skip`) and the map M denoting the heap is updated accordingly. We call such pairs $\varphi \Rightarrow \varphi'$ of matching logic formulae **reachability rules**. As another example, the reachability rule

$$\langle \texttt{skip}; stmt, M \rangle \Rightarrow \langle stmt, M \rangle$$

denotes the semantics of sequential composition.

Definition 3. *A **matching logic semantics** for a programming language is a tuple $(Cfg, S, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})$, where S is a set of sorts (denoting the syntactic categories of the language syntax), (Σ, Π) is an S-sorted first-order signature denoting the constructs of the language, $Cfg \in S$ is the sort of **configurations** and $\mathcal{T}$ a (Σ, Π)-model.*

The set of reachability rules $\mathcal{A}$ contains all rules that define the semantics of the language and $\rightarrow_{\mathcal{T}}$ is the transition system generated by $\mathcal{A}$ on $\mathcal{T}$, i.e., $\rightarrow_{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ with $\gamma \rightarrow_{\mathcal{T}} \gamma'$ iff there exist $\varphi \Rightarrow \varphi' \in \mathcal{A}$ and ρ such that $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$.

As an example, the sets $\mathcal{A}_{\mathtt{I}}$ (resp. $\mathcal{A}_{\mathtt{I}}$) of reachability rules in Figures 1 and 2 define a classic functional programming language `FUN` and a classic imperative programming language `IMP`.

2 Language Aggregation

Let $(Cfg_i, S_i, \Sigma_i, \Pi_i, \mathcal{T}_i, \mathcal{A}_i, \rightarrow_{\mathcal{T}_i})$, $i \in \{L, R\}$ be the matching logic semantics of two languages, $(S_0, \Sigma_0, \Pi_0, \mathcal{T}_0)$ the common parts of the languages L and R, h_L and h_R morphisms embedding the common part into the two languages.

The construction of the aggregated language $(Cfg, S, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})$ of the two definitions is informally presented by the diagram in Figure 3. The triple $(h'_L, (S', \Sigma', \Pi'), h'_R)$ is the pushout of $(S_L, \Sigma_L, \Pi_L) \xleftarrow{h_L} (S_0, \Sigma_0, \Pi_0) \xrightarrow{h_R} (S_R, \Sigma_R, \Pi_R)$ in the category of the first-order signatures (see [3] for details). The pushout provides a signature that encompasses the signatures of the L language and the R language. The (S', Σ', Π')-model $\mathcal{T}'$ is is obtained by amalgamation, i.e. it is smallest model such that $\mathcal{T}'{\restriction_{h'_L}} = \mathcal{T}_L$ and $\mathcal{T}'{\restriction_{h'_R}} = \mathcal{T}_R$ (the detailed construction is given in [3]). The model $\mathcal{T}'$ will serve as a model for the aggregated language. Note that $\mathcal{T}'$ is a first-order model. The embedding morphism h builds the aggregations as follows:

- Cfg is a new distinguished sort;
- $S = S' \cup \{Cfg\}$
- $\Sigma = \Sigma' \cup \{\langle \_, \_ \rangle : h_L(Cfg_L) \times h_R(Cfg_R) \rightarrow Cfg, pr_i : Cfg \rightarrow Cfg_i,\ i \in \{L, R\}\}$;
- $\Pi = \Pi'$;
- $\mathcal{T}_{Cfg} = {\mathcal{T}'}_{h'_L(Cfg_L)} \times {\mathcal{T}'}_{h'_R(Cfg_R)}$
- $\mathcal{T}_{\langle \_, \_ \rangle}(\gamma_L, \gamma_R) = (\gamma_L, \gamma_R)$, $\mathcal{T}_{pr_L}((\gamma_L, \gamma_R)) = \gamma_L$, $\mathcal{T}_{pr_R}((\gamma_L, \gamma_R)) = \gamma_R$.
- $\mathcal{T}_o = {\mathcal{T}'}_o$ for any other object $o \in S \cup \Sigma \cup \Pi$.
- $\mathcal{A} = \{\langle \_, \varphi \rangle \Rightarrow \langle \_, \varphi' \rangle \mid \varphi \Rightarrow \varphi' \in \mathcal{A}_R\} \cup \{\langle \varphi, \_ \rangle \Rightarrow \langle \varphi', \_ \rangle \mid \varphi \Rightarrow \varphi' \in \mathcal{A}_L\}$.

$$\begin{array}{ccccc}
 & & (S_0, \Sigma_0, \Pi_0, \mathcal{T}_0) & \xrightarrow{h_R} & (Cfg_R, S_R, \Sigma_R, \Pi_R, \mathcal{T}_R, \mathcal{A}_R, \rightarrow_{\mathcal{T}_R}) \\
 & & \downarrow{\scriptstyle h_L} & & \downarrow{\scriptstyle h'_R} \\
(Cfg_L, S_L, \Sigma_L, \Pi_L, \mathcal{T}_L, \mathcal{A}_L, \rightarrow_{\mathcal{T}_L}) & \xrightarrow{h'_L} & (S', \Sigma', \Pi', \mathcal{T}') & \xrightarrow{h} & (Cfg, S, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})
\end{array}$$

Fig. 3. Push-out diagram assumed throughout the paper.

Note that in the aggregated model, sorts such as integers, strings, booleans, etc that are common in both languages are interpreted by the same set and that a pair of programs takes a step in the aggregated language whenever an individual program takes a step in the initial languages.

3 Specifying Equivalent Programs

ML formulae over the aggregated language can be used to specify equivalence. According to Definition 2, the denotation of an aggregated matching logic formula φ, written $[\![\varphi]\!]$, is the set of all pairs of configurations that satisfy it:

$$[\![\varphi]\!] = \{(\langle \gamma_L, \gamma_R \rangle \mid \text{there exists a valuation } \rho \text{ such that } (\langle \gamma_L, \gamma_R \rangle, \rho) \models \varphi\}.$$

This extends to sets E of formulae ($[\![E]\!]$), as expected:$[\![E]\!] = \cup_{\varphi \in E} [\![\varphi]\!]$.

Example 1. The following set

$$E = \{\exists i. \langle \langle \texttt{skip}, (\texttt{x} \mapsto i, \_) \rangle, \langle j \rangle \rangle \wedge i =_{Int} j\}$$

containing one matching logic formula, captures in its denotation all pairs of `IMP` and respectively `FUN` (a functional language) configurations that have terminated (since there is no more code to execute) and where the `IMP` variable `x` holds the same integer as the result of the `FUN` program. Note that in the above pattern, _ is an anonymous variable meant to capture all of the variable bindings other than `x`.

Suppose we have an `IMP` program that computes its result in a variable `x` and suppose we want to show it computes the same integer result as a `FUN` program. Then the denotation $[\![E]\!]$ of set E above holds exactly the set of pairs of terminal configurations in which the two programs should end in order for them to compute the same result.

The aggregated language is useful, for example, to prove equivalence of programs. Suppose that we have a pair of programs $\langle P_L, P_R \rangle$ that both compute the sum of the first n natural numbers:

$S_L \equiv$ `s := 0; for i := 1 to n do s := s + i`
$S_R \equiv$ `let sum i = if i == 0 then 0 else i + sum (i - 1) in sum n`

The first program is written in `IMP` and the second in `FUN`. We can prove that P_L and P_R are equivalent (i.e. that they compute the same result) by showing a partial correctness property, namely that the pair $\langle P_L, P_R \rangle$ reaches the set E of final configurations defined in the previous section in any execution of the aggregated program:

$$\langle P_L, P_R \rangle \Rightarrow^* E.$$

Showing partial correctness over the aggregated language can be done using bounded model symbolic checking [2], or reachability logic [5], or Hoare-like logics over the aggregated language [1]. Based on the aggregated language, we have also developed a method of proving mutual equivalence [3].

4 Conclusion

In this talk we present in detail the construction of the aggregated language, its properties, and we discuss how various definitions of program equivalence can be defined and proved using the aggreagation. Impolementations of the aggregation operation and of the proof system for the mutual equivalence [3] is in progress. We expect that a demo will be possible during the workshop.

References

1. A. Arusoaie. Engineering hoare logic-based program verification in k framework. In *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, IEEE, 2013.
2. A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301, October 2013.
3. Ştefan Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu. A Language-Independent Proof System for Mutual Program Equivalence (revisited). Technical Report TR 14-01, UAIC Iaşi, 2014.
4. G. Roşu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
5. G. Rosu, A. Stefanescu, Ştefan Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS*, pages 358–367, 2013.