Normalization of Extended Abstract State Machines

Cégielski,Patrick
Université Paris Est–IUT, LACL EA 4219,
Route forestière Hurtault, F-77300 Fontainebleau, France,
e-mail: cegielski@univ-paris12.fr

Guessarian,Irène
LIAFA, UMR 7089 and Université Paris 6,
2 Place Jussieu, 75254 Paris Cedex 5, France;
corresponding author
e-mail: ig@liafa.jussieu.fr

ABSTRACT

We compare the control structure of Abstract State Machines (in short ASM) defined by Yuri Gurevich and AsmL, a language implementing it. AsmL is not an algorithmically complete language, as opposed to ASM, but it is closer to usual programming languages because it allows `until` and `while` iterations and sequential composition. We here give a formal definition of AsmL, its semantics, and we construct, for each AsmL program Π, a normal form (which is an ASM program Π_n) computing the same function as Π.
Keywords:Abstract State Machine, algorithm.

1. INTRODUCTION

Yuri GUREVICH has given a schema of languages which is not only a *Turing-complete language* (a language allowing to express at least an algorithm for each computable function), but which also allows to express all algorithms for each computable function (it is an *algorithmically complete language*); this schema of languages was first called *dynamic structures*, then *evolving algebras*, and finally *ASM* (for *Abstract State Machines*) [4]. He proposed the Gurevich's thesis (*the notion of algorithm is entirely captured by the model*) in [5].

There exist several partial implementations of ASMs as a programming language. These implementations are *partial* by nature for two reasons: (i) ASMs allow to program functions computable by Turing machines with oracles, but obviously not the implementations; (ii) in ASMs any (computable) first order structures may be defined, which is not true for the current implementations.

Yuri GUREVICH has directed a group at Microsoft Laboratories (Redmond) which has implemented such a programming language, called *AsmL* (for *ASM Language*), written first in C++ then in C# as a language of the .NET framework of Microsoft. To invite programmers to use its language, this group extended the control structures used in ASMs. Pure ASM control structures are represented by **normal forms** of AsmL programs.

The aim of this paper is to give a formal definition of AsmL (more precisely of the part concerning control structures; we are not interested by constructions of first-order structures here), a formal definition of normal forms in AsmL and to show how to build a normal form from an AsmL program.

2. DEFINITIONS

2.1 Definition of ASMs

We first recall the definition of ASMs and then precise our point of view on ASMs because different definitions exist.

2.2 Syntax

Definition 1. An **ASM vocabulary***, or* **signature***, is a first-order signature $\mathcal{L}$ with a finite number of function symbols, two boolean constant symbols (***true** *and* **false***), a constant symbol (denoted by* **null** *or* **undef***), logical connectives (***not***,* **and***, and* **or***), and the* **equality** *predicate.*

Terms are defined as usual in first order logic (see for instance [1]). Let us note that an ASM signature is a first-order signature whose only predicate symbol is **equality**. The fact that there are no other relation symbols simplifies some definitions and proofs but is not essential.

Definition 2. **Boolean terms** *(or* **formulæ***) are defined inductively by:*

– if t and t' are terms, then $t = t'$ is a formula;

– if p is a functional symbol onto $\{true, false\}$ of arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula;

– if F, F' are formulæ, then $\neg F$, $F \wedge F'$, $F \vee F'$ are formulæ.

Definition 3. Let $\mathcal{L}$ be an ASM signature. ASM rules are defined inductively as follows:

- *An* **update rule** *is an expression of the form:*

  $$f(t_1, \ldots, t_n) := t_0$$

  where f is a n-ary functional symbol and t_0, $t_1, \ldots, t_n$ are closed terms of $\mathcal{L}$.

- *If $R_1, \ldots, R_k$ are rules of signature $\mathcal{L}$, then the following expression is also a rule of $\mathcal{L}$, called* **block***:*

  par
  $\quad R_1$
  $\quad \vdots$
  $\quad R_k$
  endpar

  If $k = 0$, it is the empty instruction, denoted **skip***.*

- *Let φ be a boolean term, and let R_1 and R_2 be rules of signature $\mathcal{L}$. The expression:*

  if φ then R_1
  else R_2
  endif

  is a rule of vocabulary $\mathcal{L}$, called **alternative rule***.*

  The expression if φ then R_1 is a rule called **test rule***.*

At the beginning of ASMs, *par* meant *parallel*, nowadays it stands for *parenthesis*.

Definition 4. *Let $\mathcal{L}$ be an ASM signature. A* **program** *on signature $\mathcal{L}$, or $\mathcal{L}$-***program** *is a synonym for a rule of that signature.*

2.3 Semantics

Definition 5. *If $\mathcal{L}$ is an ASM signature, an ASM* **abstract state***, or more precisely an $\mathcal{L}$-***state***, is a synonym for a first-order structure $\mathcal{A}$ of signature $\mathcal{L}$ (an $\mathcal{L}$-structure).*

The universe of $\mathcal{A}$ consists of the disjoint union of three sets: the *basis set* A, the Boolean set $\mathbb{B} = \{true, false\}$, and a singleton set $\{\perp\}$. The values of the Boolean constant symbols **true** and **false** and of **null** (or **undef**) in $\mathcal{A}$ will be denoted by *true*, *false* and *null* (or $\perp$).

Definition 6. *Let $\mathcal{L}$ be an ASM signature and A a non empty set. A* **set of modifications** *(more precisely an* **($\mathcal{L}$, A)-modification set***) is any finite set of triples:*

$$(f,\ \overline{a},\ a),$$

where f is a function symbol of $\mathcal{L}$, $\overline{a} = (a_1, \dots,\ a_n)$ is an n-tuple of A (where n is the arity of f), and a is an element of A.

Definition 7. *Let $\mathcal{L}$ be an ASM signature, let $\mathcal{A}$ be an $\mathcal{L}$-state and let Π be an $\mathcal{L}$-program. Let $\Delta_\Pi(\mathcal{A})$ denote the set defined by structural induction on Π as follows:*

1. *If Π is an update rule: $f(t_1, \dots,\ t_n) := t_0$, then, denoting $a_1 = \overline{t_1}^{\mathcal{A}}, \dots,\ a_n = \overline{t_n}^{\mathcal{A}}$ and $a = \overline{t_0}^{\mathcal{A}}$, the set $\Delta_\Pi(\mathcal{A})$ is the singleton:*

   $$\{(f,\ (a_1, \dots,\ a_n),\ a)\}$$

2. *If Π is a block: par $R_1 \dots\ R_n$ endpar , then the set $\Delta_\Pi(\mathcal{A})$ is the union:*

   $$\Delta_{R_1}(\mathcal{A}) \cup \dots \cup \Delta_{R_n}(\mathcal{A})$$

   for $n \neq 0$ and the empty set otherwise.

3. *If Π is a test: if φ then R , we first have to evaluate the expression $\overline{\varphi}^{\mathcal{A}}$. If it is false then the set $\Delta_\Pi(\mathcal{A})$ is empty, otherwise it is equal to:*

   $$\Delta_R(\mathcal{A}).$$

   The semantics of the alternative rule is similar.

We may check that $\Delta_\Pi(\mathcal{A})$ is an ($\mathcal{L}$,A)-set of modifications.

Definition 8. *A set of modifications is* **incoherent** *if it contains two elements $(f,\ \overline{a},\ a)$ and $(f,\ \overline{a},\ b)$ with $a \neq b$. It is* **coherent** *otherwise.*

Definition 9. *Let $\mathcal{L}$ be an ASM signature, Π an $\mathcal{L}$-program, and $\mathcal{A}$ an $\mathcal{L}$-state.*

If $\Delta_\Pi(\mathcal{A})$ is coherent, the **transform** *$\tau_\Pi(\mathcal{A})$ of $\mathcal{A}$ by Π is the $\mathcal{L}$-structure $\mathcal{B}$ defined by:*

- *the base set of $\mathcal{B}$ is the base set A of $\mathcal{A}$;*
- *for any element f of $\mathcal{L}$ and any element $\overline{a} = (a_1, \dots,\ a_n)$ of A^n (where n is the arity of f):*
  - *if $(f,\ \overline{a},\ a) \in \Delta_\Pi(\mathcal{A})$ for an $a \in A$, then :*

    $$\overline{f}^{\mathcal{B}}(\overline{a}) = a\ ;$$

  - *otherwise:*

    $$\overline{f}^{\mathcal{B}}(\overline{a}) = \overline{f}^{\mathcal{A}}(\overline{a}).$$

If $\Delta_\Pi(\mathcal{A})$ is incoherent then $\tau_\Pi(\mathcal{A}) = \mathcal{A}$ (hence the state is a fixed point).

Definition 10. *Let $\mathcal{L}$ be an ASM signature, Π an $\mathcal{L}$-program, and $\mathcal{A}$ an $\mathcal{L}$-state. The* **computation** *is the sequence $(\mathcal{A}_n)_{n \in \mathbb{N}}$ defined by:*

- *$\mathcal{A}_0 = \mathcal{A}$ (called the* **initial algebra** *of the computation);*
- *$\mathcal{A}_{n+1} = \tau_\Pi(\mathcal{A}_n)$ for $n \in \mathbb{N}$.*

A computation **terminates** *if there exists a fixed point $\mathcal{A}_{n+1} = \mathcal{A}_n$.*

Definition 11. *The* **semantics** *is the partial function which transforms $\{\Pi, \mathcal{A}\}$, where Π is an ASM program and $\mathcal{A}$ a state, in the result of iterating τ_Π starting from $\tau_\Pi(\mathcal{A})$ until a fixed point is reached (it will be denoted $[\![(\Pi, \mathcal{A})]\!] = \tau_\Pi^*(\mathcal{A})$) if such a fixpoint exists, otherwise it is undefined.*

3. DEFINITION OF EXTENDED ASMS

We call *extended ASM* the analog of ASM with additional rules to take into account the supplementary control structures of AsmL. No paper defines formally AsmL but [3] gives a precise informal description.

3.1 Syntax

Roughly speaking, AsmL considers more control structures than ASMs. We first define AsmL rules, then explain them.

Definition 12. *Let $\mathcal{L}$ be an ASM signature.*

- *An ASM rule is also an* **extended rule** *of $\mathcal{L}$.*
- *If φ is a boolean term, and R_1 and R_2 are extended rules, then:*

  if φ then R_1
  else R_2
  endif

  is an extended rule, called **alternative rule***.*
- *If $R_1, \dots,\ R_k$ are extended rules of signature $\mathcal{L}$, then the following expression is also an extended rule of $\mathcal{L}$, called* **step rule***:*

  par
  step R_1
  ⋮
  step R_k
  endpar

- *If R is an extended rule of signature $\mathcal{L}$, then the following expressions are also extended rules of $\mathcal{L}$, called* **iteration rules***:*
  - **step until** φ R
  - **step while** φ R
  - **step until fixpoint** R

  with φ a boolean term.

We now explain the above rules. Following the semantics given above for ASMs, the meaning of the **par** rule:

par
    R_1
    $\vdots$
    R_k
endpar

is that rules R_1, ..., R_k are running simultaneously, i.e. each rule is running (with a point of view sequential or parallel for the observer) independently of the other rules; incoherences might occur for some updates; if there is at least one incoherence, the program stops, otherwise updates are applied.

The paradigm of simultaneity is unusual for programmers who prefer sequentiality. Hence sequentiality was introduced using **step** in AsmL. The meaning of the extended rule

par
    $step_1$
    $\vdots$
    $step_k$
endpar

is as follows: rule $step_1$ is running first; then rule $step_2$ is running (with values of closed terms depending of the result of running rule $step_1$); and so on.

Finally classical iterations appear in AsmL via the **step until** and **step while iteration rules**, and the **step until fixpoint** rule of ASM is also present.

4. NORMALIZATION

Definition 13. *Let $\mathcal{L}$ be an ASM vocabulary. If R is an extended rule of vocabulary $\mathcal{L}$, then the expression* **step until fixpoint** R *is called a* **running rule**.

Running rules are implicit in ASMs and are made explicit in AsmL. If R is an ASM rule (not extended), then **step until fixpoint** R will have the same semantics as the ASM program Π consisting of rule R.

Definition 14. *Let $\mathcal{L}$ be an ASM vocabulary. If R_1, ..., R_k are ASM rules of vocabulary $\mathcal{L}$ (*not extended rules*), then the extended rule:*

```
step  until   fixpoint
      par
              if (mode = 0) then R_1
              ...
              if (mode = j) then R_j
              ...
              if (c=0) ∧ (mode = i) then R_i
              if (c=1) ∧ (M_i ≥ mode ≥ i) then
                  mode:=i
                  c:=0
              ...
              if (mode = k) then R_k
      endpar
```

is a **normal form** *of $\mathcal{L}$, where $mode, c$ are special nullary functional constants of the ASM language whose value for the initial algebra are 0.*

The rule under `step until fixpoint` *is called the* **core** *of the normal form.*

In the sequel `par` and `endpar` will be omitted: when rules have the same indentation, they will be assumed to be in a `par...  endpar` block.

To give a normal form for a given program is a classical issue of theoretical computer science.

Example Consider the following programming problem: to compute the average of an array of grades and to determine the number of grades whose value is greater than this average. A natural program in extended ASMs is:

```
step
        avg := 0
        i := 0
step    until (i = n)
        avg := avg + grade[i]
        i := i+1
step
        avg := avg/n
        i := 0
        nb := 0
step    until (i = n)
        if (grade[i] > avg) then nb := nb+1
        i := i+1
```

using the conventions of AsmL (a same indentation indicates a block).

The core of the normal form is:

```
if   (mode = 0) then
     avg := 0
     i := 0
     mode := 1
if   (mode = 1) then
     avg := avg + grade[i]
     i := i+1
     if (i = n) then mode := 2
if   (mode = 2) then
     avg := avg/n
     i := 0
     nb := 0
     mode := 3
if   (mode = 3) then
     if (grade[i] > avg) then nb := nb+1
     i := i+1
     if (i = n) then mode := 4
if   (mode = 4) then
     skip
```

Theorem 15. *For every extended ASM program Π, there exists a normal form ASM program Π_n such that for every $\mathcal{L}$-state $\mathcal{A}$ whose base set is infinite there exists an $\mathcal{L}_m$-state $\mathcal{A}_m$ in which Π_n computes the same function as Π.*

Proof. We define the core of the normal form Π_n of Π by structural induction on the form of program Π. The proof that Π_n and Π compute the same function can be found in [2].

We will need to label some blocks. To allow this, we suppose the base set is infinite, and we use special nullary functions, called *mode*, c and whose corresponding constant symbols are added to the ASM language, i.e. $\mathcal{L}_m = \mathcal{L} \cup \{mode, c\}$. Without loss of generality, we suppose the set $\mathbb{N}$ of natural integers is included in the base set.

- The core of the normal form R_n of an ASM rule R is:

  if (mode = 0) then
  R
  mode := 1

- If $R'_1, \ldots, R'_k$ are the respective cores of the extended rules $R_1, \ldots, R_k$, we have to define the core of the extended rule Π:

  par
  step R_1
  ...
  step R_k
  endpar

  In the (abnormal) case $k = 1$, the core is simply R'_1. To explain the case $k \geq 2$, it is sufficient to suppose $k = 2$. In this case the core of Π_n is:

  R''_1
  $R_2^{+fin_1}$

  with

  1. $R''_1 = R'_1$.
  2. $R_2^{+fin_1}$ is defined as follows: Let fin_1 be the greatest value used for *mode* in R'_1. Rule $R_2^{+fin_1}$ is R'_2 where fin_1 is added to each *constant* occurring on the right side of the "=" (or "≥") sign of an expression rule beginning by *mode* (a boolean expression *mode* = *constant* or an update rule *mode* := *constant*).

- If Π is an iteration rule (`step until` φ R) and R' is the core of the normal form of R then the core Π' of the normal form Π_n of Π is

  if (mode = 0) then
  if $\neg\varphi$ then mode := 1
  if φ then mode := $fin + 1$
  R^{+1}

  where R^{+1} is R' in which each *constant* occurring on the right side of the "="(or "≥") sign of an expression beginning by *mode* (a boolean expression *mode* = (≥)*constant* or an update rule *mode* := *constant*) is incremented by 1 but for the greatest such value fin, which is replaced by 0.

- If Π is an iteration rule (`step until fixpoint` R), then the stopping condition of the iteration is defined as follows in [3] *"In the case of until fixpoint, the stopping condition is met if no non-trivial updates have been made in the step. Updates that occur to variables declared in abstract state machines that are nested inside the fixed-point loop are not considered. An update is considered non-trivial if the new value is different from the old value."* We thus must check that all updates are trivial before stopping. Let $f_i(t_1, \ldots, t_n) := t_0^i$, $i = 1, \ldots, k$, be all the (non nested) updates performed in R. Let us introduce a new constant c (keeping track of non trivial updates), with value 0 in the initial algebra.

  Let R' be the core of the normal form of R and M the largest value of mode occurring in R', , then the core Π' of the normal form Π_n of Π is:

  if (c=1) $\wedge$ ($M \geq$ mode ≥ 0) then
  mode:=0
  c:=0
  if (c=0) $\wedge$ (mode = 0) then R^c

  R^c is R' in which each update (other than updates on c and *mode*) $f_i(t_1, \ldots, t_n) := t_0^i$ has been replaced by

  if $f_i(t_1, \ldots, t_n) \neq t_0^i$ then
  $f_i(t_1, \ldots, t_n) := t_0^i$
  $c := 1$

- If Π is an iteration rule (`step while` φ R), it is treated as the until iteration, and the core of the normal form is:

  if (mode = 0) then
  if φ then mode := 1
  if $\neg\varphi$ then mode := $fin + 1$
  R^{+1}

- If Π is an **alternative rule**,
  if φ then R_1
  else R_2
  endif
  and R'_1, R'_2 are the respective cores of R_1, R_2, then the core of the normal form of Π is:

  if (mode = 0) then
  if φ then mode := 1 else mode := $fin + 1$
  R_1^{+1}
  R_2^{fin+1}

  where R_i^{+k} is R'_i where k is added to each *constant* occurring on the right side of the "=" (or "≥") sign of an expression rule beginning by *mode*.

□

REFERENCES

[1] ARNOLD, André, GUESSARIAN, Irène, *Mathematics for Computer Science*, Prentice-Hall, London (1996), 401 p.

[2] CEGIELSKI, Patrick, GUESSARIAN, Irène, *Normal Forms for Extended Abstract State Machines*, `http://www.liafa.jussieu.fr/~ig/g70.ps`

[3] GRIESKAMP, Wolfgang, TILLMANN, Nikolai, **AsmL Standard Library**, Foundations of Software Engineering – Microsoft Research, 2002, 38 p., see also `http://www.codeplex.com/AsmL//AsmLReference.doc` `http://research.microsoft.com/en-us/downloads/3444a9cb-47ce-4624-9e14-c2c3a2309a44/default.aspx`

[4] GUREVICH, Yuri, *Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs*, University of Michigan, Technical Report CRL–TR–38–84, EECS Department, 1984.

[5] GUREVICH, Yuri, *A New Thesis*, **Abstracts, American Mathematical Society**, August 1985, p. 317.