A Modular Implementation of Action Notation

Leonardo Mendonça de Moura
e-mail: moura@les.inf.puc-rio.br

Carlos José Pereira de Lucena
e-mail: lucena@inf.puc-rio.br

Edward Hermann Haeusler
e-mail: hermann@inf.puc-rio.br

PUC-RioInf.MCC07/00 February, 2000

Abstract: This paper describes the implementation of a modular interpreter for the new modular semantics of action notation, the formal notation used in *action semantics*. Our interpreter supports nondeterministic features of action semantics, and allows the user to explore different execution traces. The implementation is based on a new logic programming language called PAN. A brief and pragmatic introduction to modular structured operational semantics is also provided.

Keywords: Action Semantics, Modular SOS, Logic Programming.

Resumo: Este artigo descreve a implementação de um interpretador modular para a nova semântica modular da notação de ações, i.e. a notação formal utilizada em semântica de ações. Nosso interpretador suporta não determinismo, e permite que o usuário explore diferentes *traces* de execução. A implementação é baseada em uma nova linguagem lógica denominada PAN. Uma breve introdução a SOS Modular também é apresentada.

Palavras-chave: Semântica de Ações, SOS Modular, Programação Lógica.

1 Introduction

We present here an implementation of a modular interpreter for the new modular semantics of action notation, the formal notation used in *action semantics* [7]. Such implementation involves the use of a new logic programming language called PAN which we specially developed to implement language interpreters, analysis and verification tools. PAN has several features which are extremely useful for specifying the operational semantics of programming languages. An interpreter for action notation is particularly interesting, since it allows us to experiment in the early stages of the language design, which is very useful to consolidate our understanding of the language design.

Action Semantics [7] is a framework for the formal description of programming languages. Not only denotational semantics [15, 19], but also action semantics are compositional, *i.e.* the semantics of each phrase is determined by the semantics of its subphrases. The difference is that the semantics of phrases are *actions* instead of higher-order functions, which are commonly used in denotational semantics descriptions. Thus, action semantics might be regarded as denotational, where the denotations are actions. Action Notation, the notation used in action semantics descriptions, has an operational semantics, and nice algebraic properties [7]. The main concept in action semantics is the concept of action.

In structural operational semantics (SOS) [12] literature modularity issues have largely been neglected, and SOS descriptions of programming languages typically exhibit rather poor modularity. Thus, the original operational semantics of action notation was not modular. Therefore, we would not be able to reuse the existing semantic description when extending or changing the action notation. Recently, Mosses described how to obtain a high degree of modularity in SOS [8], and the SOS of Action Notation is currently being reformulated in the proposed modular style.

2 A Brief Introduction to MSOS

In this section we provide a brief introduction to the method of *Modular Structural Operational Semantics* (MSOS), which is a modular variation of SOS. SOS represents computations by means of deductive systems, transforming an abstract machine into a system of logical inferences. SOS definitions are given by inference rules consisting of conclusions that follow from a set of premises. Such rules describe possible transitions (*small steps*) of a computation. Each proof in the deductive system describes a state transition.

SOS involves abstract syntax, computed values, configurations and inference rules for (labelled) transitions. A SOS description specifies a *labelled transition system* $(\Gamma, T, A, \rightarrow)$, where Γ is a set of configurations, $T \subseteq \Gamma$ is a set of terminal configurations, A is a set of labels, and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is a transition relation. For configuration $\gamma, \gamma' \in \Gamma$ and labels $\alpha \in A$, the assertion that $(\gamma, \alpha, \gamma')$ is in the transition relation is written as $\gamma \xrightarrow{\alpha} \gamma'$.

On the other hand, for our objectives SOS descriptions can be seen as interpreters implemented in a logic programming language. The main predicate of these logical programs is the one which defines the transition relation.

However, most of the SOS descriptions lack modularity. Local modifications

$$skip \xrightarrow{i} completed \qquad \frac{set\text{-}store(\alpha, Id, Val)}{Id := Val \xrightarrow{\alpha} completed}$$

$$\frac{Be \xrightarrow{\alpha} Be'}{if\ Be\ then\ S_1\ else\ S_2 \xrightarrow{\alpha} if\ Be'\ then\ S_1\ else\ S_2}$$

$$if\ true\ then\ S_1\ else\ S_2 \xrightarrow{i} S_1 \qquad if\ false\ then\ S_1\ else\ S_2 \xrightarrow{i} S_2$$

$$while\ E\ do\ S \xrightarrow{i} if\ E\ then\ S; while\ E\ do\ S\ else\ skip$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} S_1'; S_2} \qquad completed\ ; S_2 \xrightarrow{i} S_2$$

Figure 1: Modular SOS example

in language semantics usually imply global modifications in SOS descriptions. This also means that it is almost impossible to *reuse* SOS descriptions when defining a new language. To cope with this problem, we use an approach similar to the one defined in [8]. In this approach, the modularity is achieved by using encapsulation techniques. Figure 1 shows a small fragment of a modular SOS description for a simple imperative language. In this example, the configurations are restricted to abstract syntax trees, where nodes may be replaced by the values they computed, as in conventional SOS. The initial configurations are pure syntax, and terminal configurations are simply computed values. Value *completed* represents normal termination. All usual semantic components of configurations, such as stores, are incorporated into the labels on transitions. All labels contain a pair of stores reflecting the value of the store before and after the transition. The structure of the label is encapsulated, and should only be accessed by means of predicates. We can view the label as an object [1] and the predicates as the interface (methods) of the object.

All labels contain a partial composition predicate used to assert whether two transitions may be adjacent. In the example in Figure 1, this predicate is defined as:

$$compose(\alpha_1, \alpha_2, \alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha_1 & = & \langle store_1, store_2 \rangle \wedge \\ \alpha_2 & = & \langle store_2, store_3 \rangle \wedge \\ \alpha & = & \langle store_1, store_3 \rangle \end{cases}$$

This predicate asserts that label α is the composition of labels α_1 and α_2. Notice that in [8] such predicate is represented by the operation _;_, thus we may seen $\alpha = \alpha_1; \alpha_2$ as a syntax sugar for $compose(\alpha_1, \alpha_2, \alpha)$. Every label can always be composed on the left and on the right with a specific identity label. The notion of identity labels is defined by the predicate *id*. In the given example, this predicate is defined as:

$$id(\alpha) \stackrel{\text{def}}{=} (\alpha = \langle store, store \rangle)$$

[1] Using the object-oriented terminology.

We use $\gamma \xrightarrow{i} \gamma'$ as a syntax sugar for:

$$\frac{id(\alpha)}{\gamma \xrightarrow{\alpha} \gamma'}$$

As already mentioned, all labels should provide predicates to access their internal structure. In the given example, the label contains two predicates:

$$\begin{aligned} \textit{set-store}(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ S_2 = S_1[Id \leftarrow Val] \\ \textit{get-store}(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ Val = S_1(Id) \end{aligned}$$

The predicate *set-store* "modifies" the value of a given variable in the store. The predicate *get-store* "accesses" the value of a given variable.

Using this approach, we do not need to completely modify a SOS description when we extend or modify the language semantics.

3 The *PAN* Programming Language

In our implementation, the operational semantics of a language must be specified in a programming language called *PAN*. PAN is a logic programming language specially developed to specify interpreters, program analyzers and verifiers. The PAN syntax and semantics have some similarities with *Prolog* [18]. The PAN compiler uses global analysis to produce code with a performance comparable to the one produced by imperative language compilers.

All PAN programs must be properly qualified by providing the kind, type and mixfix declarations. For instance, consider the following specification of the concatenation of lists in PAN syntax:

```
append nil X X.
append (X :: Xs) Ys (X :: Zs) :- append Xs Ys Zs.
```

This specification contains four variables `X`, `Xs`, `Ys` and `Zs`, one logical constant, namely `:-` for the converse of implication, and three non-logical constants, namely `nil`, `::` and `append`, denoting the empty list, the list constructor, and the concatenation relation, respectively. Two of these constants, namely `:-` and `::` are also used as infix symbols. This specification is only meaningful if mixfix declarations and type declarations are given for these constants.

```
type nil : list A.
type _ :: _ : A -> list A -> list A {prec 8 r-assoc}.
type append : list A -> list A -> list A -> o.
```

PAN has a polymorphic type system similar to the type system of ML [6]. In type declarations, tokens with an initial upper case letter denote *type variables*. The type declaration for `nil` above asserts that for every type `A`, the constant `nil` is of type `(list A)`. The type declaration for `_ :: _` asserts that given a term `H` of type `A`, and a term `T` of type `list A`, then the term `T :: H` is of type `list A`. If the target type of a type declaration is `o`, then the constant is a predicate symbol. In the above example, `append` is a predicate symbol that contains three arguments.

The underbars in constants like `_ :: _` are place holders that indicate where terms of type `A` and `list A` should go, respectively. The precedence attribute `prec` n tells how tightly a symbol binds. The higher the number the tighter the symbol binds. The attributes `r-assoc` and `l-assoc` tell if a symbol is right or left associative. Precedence and associativity information is used to disambiguate the parsing. Notice that the above declarations have introduced two new constants not in the original specification, namely `o` and `list`. These type constants will need declarations, called *kind declarations*. Kind declarations are used to introduce *type constructors*. An example of kind declaration is:

```
kind list : type -> type.
```

Here, `list` takes a type and returns another type. For example, `list (list int)` is the type of a list of lists of integers.

PAN also supports higher orders predicates, i.e. predicates that have other predicates as arguments. For example:

```
type map : (A -> B -> o) -> list A -> list B -> o.
map P nil nil.
map P (X :: Xs) (Y :: Ys) :- (P X Y), map P Xs Ys.
```

The goal `(map F Xs Ys)` is provable, if `Xs` and `Ys` are lists of equal length and corresponding elements of these lists are related by the predicate `P`.

Prolog has various all-solutions predicates (e.g. findall/3), all of them having semantic problems. PAN has the builtin predicate called *solutions*. We avoid the variable scoping problems of most Prolog versions. Rather than taking a goal to execute and an aliased term holding the resulting value to collect, the predicate *solutions* take as input a single higher-order predicate. The declaration of the predicate *solutions* is:

```
type solutions _ _  : ((A0 -> o) -> (list A0) -> o)
```

PAN compiler is based on state of the art logic programming compilation technology [13, 17]. The compiler uses several global analysis (mode analysis, determinism analysis, switch detection, etc) to produce efficient code. It is out of scope of this article to explain each one of such analysis algorithms. The output of the PAN compiler is portable C code. Hence, the PAN compiler allows us to invoke C code from PAN code and vice-versa.

PAN does not have "impure" features like Prolog [2]. We avoid the efficiency problems by using an approach similar to the one found in pure functional languages [20]. Briefly, the user can decorate the code with uniqueness marks. Such marks tell the compiler that a given argument of a given predicate can only have one variable referencing it. As there is only one reference to a given argument, destructive updates can be safely performed. This safety condition is checked by the compiler, and if not satisfied produces a warning message. This feature is used to allow PAN code to invoke C code which performs destructive updates in a safe way. Figure 2 shows an example of interface between PAN and C code. The first three lines show how to instruct the compiler to add arbitrary C code into the produced code. The *cdecl* declarations tell the compiler that the respective predicates are implemented by using C code. The marks *in* and *out*

[2] The only non-logical feature of the PAN language is the *cut*.

```
C{{
#include "array.h"
}}C.
kind array : type -> type.

cdecl create-empty-array : unq out array A -> o {det}
C{{ // C code ... }}C.

cdecl update-array : in int -> in A -> unq in array A ->
                     unq out array A -> o {det}
C{{ // C code ... }}C.
```

Figure 2: Interface with the C language

```
kind stmt : type.
kind expr : type.

type _ ; _ : stmt -> stmt -> stmt {prec 80 l-assoc}.
type skip : stmt.
type if _ then _ else _ end :
                 expr -> stmt -> stmt -> stmt {prec 65}.
type while _ do _ end : expr -> stmt -> stmt {prec 70}.
type _ := _ : id -> expr -> stmt {prec 100}.
```

Figure 3: Abstract syntax definition

are used to indicate which arguments are input or output. The uniqueness marks are represented by the symbol *unq*. For instance, the predicate *update-array* receives an index, a value and the array that will be updated. The resulting array is returned in the fourth argument.

The PAN features are extremely useful for specifying the operational semantics of a programming language. The mixfix notation allows us to specify the abstract syntax of a programming language and to describe the transition rules in a clear way. For instance, Figure 3 shows part of the abstract syntax definition of the simple imperative language.

Figure 4 shows part of the translation to PAN of the SOS description of Figure 1. The symbol `|-` is an alternative for `:-` that simplifies the description of SOS rules. It allows us to write a clause as `Body |- Head` instead of `Head :- Body`. The predicate `_ -- _ --> _` specifies the transition relation.

4 Implementing Action Notation

Action Notation [7] is a rich algebraic notation for expressing actions, which are used to represent the semantics of constructs of conventional programming

```
kind label : type.
kind config : type.
sub stmt : config.
type completed : config.
type _ -- _ --> _ : config -> label -> config -> o.

S1 -- L --> S1'
|-
S1 ; S2 -- L --> S1' ; S2.

id L
|-
while E do S end -- L --> if E then S ; (while E do S end) end.

(is-val E V), (set-store L ID V)
|-
ID := E -- L --> completed.
```

Figure 4: Fragment of language semantics

languages. Such notation is verbose and suggestive, which improves readability of semantic descriptions.

The performance of an action directly represents information processing behavior and reflects gradual, stepwise nature of computation: each step of an action performance may access and/or change the current information. Yielders occurring in actions may access, but not change, the current information.

A performance of an action either: completes, corresponding to normal termination; or escapes, corresponding to exceptional termination; or fails, corresponding to abandoning an alternative; or diverges.

Action notation consists of several independent parts called *facets*. AN contains the following facets:

Basic: is used to specify the flow of control in actions;

Functional: is used to specify the flow of the data;

Declarative: is used to specify the scopes of the bindings that are received and produced by actions;

Imperative: is used to specify the allocation of storage for values of variables;

Reflective: is used to specify procedural abstraction and application;

Communicative: is used to specify message passing.

In our implementation, each facet is coded in an independent PAN module. Each module requires some features from the label data structure, *i.e.* the label definition module should implement some predicates which provide

```
module action-support.

kind action : type.
kind yielder : type.
kind label : type.

// basic label "methods"
type id : label -> o.
type compose : label -> label -> label -> o.

type _ -- _ --> _ : action -> label -> action -> o.
// definition of the transitive closure of _ -- _ --> _
type _ -- _ -->+ _ : action -> label -> action -> o.
type _ -- _ --> _ : yielder -> label -> yielder -> o.

A1 -- L --> A2
|-
A1 -- L -->+ A2.

A1 -- L1 --> A2,  A2 -- L2 -->+ A3, (compose L1 L2 L)
|-
A1 -- L -->+ A3.
```

Figure 5: Fragment of the module *action-support*

the desired services. For example, the basic facet requires the predicates: *set-commitment*, *get-commitment*, *set-unfolding*, and *get-unfolding*. Notice that in the formal description of action semantics [9], each label provides the operations $set(\alpha, data, d)$ and $get(\alpha, data)$. In this way, the predicate call

$$set\text{-}commitment(\alpha, committed, \alpha')$$

used in our description is equivalent to

$$\alpha' = set(\alpha, commitment, committed)$$

The PAN module *action-support* (Figure 5) provides the basic definitions that are used in our implementation. Notice that the predicate `_ -- _ -->+ _` is the transitive closure of the predicate `_ -- _ --> _`.

The implementation of such modules is almost identical to the formal description of the facets [9]. For example, Figure 6 contains a fragment of the formal definition of the basic facet. Such fragment was extracted from [9]. As stated above, it is quite straightforward to code a SOS description in PAN. Figure 7 contains the definition of the abstract syntax tree in PAN. Figure 8 contains additional definitions. Note that we defined a new predicate called *is-terminated* which specifies if an action has been terminated or not. Figure 9 contains the implementation of the module. It is clear that such implementation is almost identical to the formal definition described in Figure 6. The implementation of the other facets is also straightforward.

type *Terminated* ::= *completed* | *escaped* | *failed*
type *Action* ::= _ **or** _(*Action*; *Action*) | **fail** | **commit** |
_ **and** _(*Action*; *Action*) | **complete** |
indivisibly _(*Action*) | _ **and then** _(*Action*;*Action*) |
unfolding _(*Action*) | **unfold**
...
sort *Terminated* | _ **@** _(*Action*; *Action*)

$$\frac{A_1 \stackrel{\alpha}{\to} A_1'}{A_1 \ \mathbf{or}\ A_2 \stackrel{\alpha}{\to} A_1' \ \mathbf{or}\ A_2} \qquad \frac{A_2 \stackrel{\alpha}{\to} A_2'}{A_1 \ \mathbf{or}\ A_2 \stackrel{\alpha}{\to} A_1 \ \mathbf{or}\ A_2'}$$

$$\mathit{completed}\ \mathbf{or}\ A_2 \stackrel{i}{\to} \mathit{completed} \qquad A_1\ \mathbf{or}\ \mathit{completed} \stackrel{i}{\to} \mathit{completed}$$

...

$$\frac{A_1 \stackrel{\alpha}{\to} A_1' \quad \mathit{get}(\alpha, \mathit{commitment}) = \mathit{committed}}{A_1\ \mathbf{or}\ A_2 \stackrel{\alpha}{\to} A_1'}$$

$$\frac{A_2 \stackrel{\alpha}{\to} A_2' \quad \mathit{get}(\alpha, \mathit{commitment}) = \mathit{committed}}{A_1\ \mathbf{or}\ A_2 \stackrel{\alpha}{\to} A_2'}$$

...

var t : *Terminated*

$$\frac{A \stackrel{\alpha}{\to}^{+} t}{\mathbf{indivisibly}\ A \stackrel{\alpha}{\to} t}$$

...

$$\mathbf{unfolding}\ A \stackrel{i}{\to} A\ \mathbf{@}\ A \qquad t\ \mathbf{@}\ A_0 \stackrel{i}{\to} t$$

$$\frac{\alpha' = \mathit{set}(\alpha, \mathit{unfolding}, A_0) \quad A \stackrel{\alpha'}{\to} A'}{A\ \mathbf{@}\ A_0 \stackrel{\alpha}{\to} A'\ \mathbf{@}\ A_0}$$

$$\frac{\mathit{get}(i, \mathit{unfolding}) = A_0}{\mathbf{unfold} \stackrel{i}{\to} A_0}$$

...

Figure 6: Fragment of the Basic Facet

```
// ----------- Abstract Syntax Tree ---------------
type _ or _ : action -> action -> action {prec 60 l-assoc}.
type _ and _ : action -> action -> action {prec 70 l-assoc}.
type indivisibly _ : action -> action {prec 50 l-assoc}.
type unfolding _ : action -> action {prec 50}.
type unfold : action.
type diverge : action.

type fail : action.
type complete : action.
type escape : action.
type commit : action.
```

Figure 7: Fragment of the abstract syntax tree definition coded in PAN

```
// values added to the AST
type _ @ _ : action -> action -> action {prec 100 l-assoc}.
type completed : action.
type failed : action.
type escaped : action.
type nothing : yielder.

// auxiliar "method"
type is-terminated : action -> o.
is-terminated completed.
is-terminated failed.
is-terminated escaped.
```

Figure 8: Additional definitions

```
A1 -- L --> A1'
|-
A1 or A2 -- L --> A1' or A2.

A2 -- L --> A2'
|-
A1 or A2 -- L --> A1 or A2'.

A1 -- L --> A1', (get-commitment L committed)
|-
A1 or A2 -- L --> A1'.

A2 -- L --> A2', (get-commitment L committed)
|-
A1 or A2 -- L --> A2'.

id L, (set-commitment L committed L')
|-
commit -- L' --> completed.

A -- L -->+ T, (is-terminated T)
|-
indivisibly A -- L --> T.

id L
|-
unfolding A -- L --> A @ A.

id L, is-terminated T
|-
T @ A -- L --> T.

(set-unfolding L A0 L'), A -- L' --> A'
|-
A @ A0 -- L --> A' @ A0.

(get-unfolding L A0)
|-
unfold -- L --> A0.
```

Figure 9: Fragment of the Basic Facet coded in PAN

The implementation of the *label* module was also quite simple. Basically, it is a tuple which stores the *transients*, *bindings* and *storage*. Such information is implemented by using *mappings*. The performance of our interpreter is improved by using the C language to implement the mapping data structure. It should be clear that the label also stores information related to commitment and unfolding, *i.e.* the predicates such as *get-commitment* and *set-unfolding* behaves like *getter* and *setter* methods, which are well known in the object-oriented literature.

Notice that was not necessary to implement a parser for action notation, since the abstract syntax definition was sufficient due to the mixfix capabilities of the PAN language. For example, if we desire to obtain the result of performing action

```
| give 2
then
| give the successor of (the given integer)
```

we should simply perform the following PAN query

```
(give 2)
then
(give the successor of (the given integer)) -- L -->+ T,
is-terminated T.
```

The label `L` contains the outcome of the computation, *i.e. transients*, *bindings* and *storage*. The user may use the predicate *display-results* to print such mappings on the screen. It is important to remember that PAN does not have side effects, therefore we use an approach similar to the one used in the Haskell [4] language to implement IO. In this way, the signature of predicate *display-results* is:

```
type display-results: unq in IO -> unq out IO -> in Label -> o.
```

The first two parameters represents the state of the "world" (IO devices) before and after executing the predicate *display-results*. The uniqueness marks allows the predicate to perform destructives updates in the "world", *i.e.* to print on the screen.

4.1 Handling the nondeterminism

It is extremely simple to describe nondeterministic and concurrent languages by using SOS, differently from denotational semantics. Action Notation provides several nondeterministic combinators such as `_or_` and `_and_`. Therefore, it is important to define how the nondeterminism will be handled by our interpreter.

We say a language is nondeterministic, when a given state of a given program written in such language has more than one successor state [3]. In other words, a given program has more than an execution trace. We define *trace* as a finite or infinite sequence of states $(s_0, s_1, \ldots, s_i, \ldots)$, where there is a transition from s_i to s_{i+1}.

Thus, one interpreter (simulator) for such kind of language must clearly state how the successor state is chosen. A naive interpreter always selects a

[3] We say s' is a successor of s if there is a transition from s to s'

```
interpret (State s0)
{
  State s = s0;
  while (successors(s) ≠ ∅)
          s = select(successors(s));
}
```

Figure 10: The "abstract" interpretation algorithm

specific trace, and it is not capable of reproducing all possible behaviors of a given nondeterministic program.

Our implementation provides infrastructure to define different "flavors" of interpretation starting from the operational semantics. Our framework supports the following "flavors" of interpretation:

Standard: the successor state is chosen deterministically. The choice is based on the order in which the rules are described.

Random Selection: the successor state is chosen randomly. For deterministic languages this "flavor" behaves as the standard interpreter.

Guided: the user chooses the successor state.

Oracle: the user provides a function that chooses the successor state. Notice that such function may request user interaction in specific points of the simulation.

Abstractly, all "flavors" described above can be seen as instances of the algorithm described in Figure 10. The only difference is the way the function *select* is implemented. It is important to notice that the all possible successors of a state is obtained by using the builtin predicate *solutions*.

If desired, the user can also select interpretation algorithms that store the selected trace during the simulation, saving the *history* of the computation. We also provide support for common debugging operations like setting breakpoints and inspecting the value of program variables. Obviously, none of such features are described in Figure 10.

4.2 Interpreting Programs

Although our interpreter may interpret arbitrary actions, it is much more useful to interpret actions which are denotations of programs. We may use two different approaches to handle such issue. The first approach uses the *actioneer generator* [10]. The actioneer generator is a program that given a semantic description of a language $\mathcal{L}$, it produces a program $\mathcal{P}$ which transforms abstract syntax trees of programs coded in $\mathcal{L}$ in their denotations (abstract syntax tree *ast* coded in action notation). In order to interpret *ast*, we must first convert it into a format of the PAN language. Fortunately, such process is straightforward.

The other approach is similar to the one described in [16]. It is based on the fact that logic programming languages are extremely useful for applying programming transformations [18]. The following semantic equations:

execute ⟦ C_1 ; C_2 ⟧ = execute C_1 and then execute C_2.

elaborate ⟦ var I : T ⟧ =
  | allocate a cell
  then
  | bind I to the given cell

are coded in PAN as:

```
type execute [[ _ ]]= _ : Command -> Action -> o.
type elaborate [[ _ ]]= _ : Declaration -> Action -> o.

execute [[ C1 ; C2 ]]= ExecuteC1 and then ExecuteC2 :-
                 execute [[ C1 ]]= ExecuteC1,
                 execute [[ C2 ]]= ExecuteC2.

elaborate [[ var I : T ]]= (allocate a cell)
                            then
                            (bind I to the given cell).
```

By using such approaches, it is possible to interpret the abstract syntax of programs coded in a language $\mathcal{L}$. A parser for language $\mathcal{L}$ may be implemented by using tools such as YACC, allowing us to implement a complete interpreter for language $\mathcal{L}$. The interpreter coded in PAN may call the generated parser coded in C.

An interpreter should provide some kind of traceability, i.e. an association between the abstract syntax tree nodes and the original source code. Such feature is important if we desire to interact with the interpreter, and to perform actions such as setting breakpoints and selecting execution traces. Thus, our implementation uses mappings to associate abstract syntax tree nodes with source code information such as line and column numbers.

5 Future Work (Partial Evaluation)

We are currently developing an automatic partial evaluator for the PAN language based on the ideas used to implement *Mixtus* [4] [14]. The absence of "impure" features in the PAN language avoids several problems found in the implementation of Mixtus.

Partial evaluation is a program transformation which specializes a program to a particular context reducing its execution time and, in some cases, its size [5]. A specialization context is defined by assigning values to some subset of a program's inputs.

Futamura [3] showed that a partial evaluator applied to an interpreter (for language X, written in a language Y) given a program will yield a new program (in language Y). This is called the first Futamura projection, and is probably the most useful and widely used aspect of partial evaluation, as the overhead of the interpreter might be eliminated.

More specifically, consider our interpreter $\mathcal{I}$ for action notation. The inputs of our interpreter is an action a and an initial state s_0 (*transients*, *bindings*, and

[4] Mixtus is an automatic partial evaluator for Prolog.

storage). Now, using partial evaluation, we obtain $\mathcal{I}_a$ which is a specialization of $\mathcal{I}$ with respect to the action a. Notice that, $\mathcal{I}_a$ is a PAN program, and our compiler should be used to convert it in an equivalent C program.

Such process may be used to produce compiled code starting from action semantics descriptions. A similar approach using Scheme is described in [1].

6 Conclusion

We described an modular interpreter for action notation which was implemented by using a new logic programming language called PAN. Such interpreter may be used to obtain action semantics executable specifications.

The interpreter has several features to handle nondeterministic actions, and it allows the user to explore different execution traces. Such feature is interesting when handling nondeterministic or concurrent programming language. As far as we know, previous interpreters and compilers [10, 1, 11, 2] for action notation are only able to handle deterministic actions, *i.e.* the nondeterministic action primitives and combinators are handled (approximated) in a deterministic way.

Due to the modular implementation, new facets may be simply introduced in our implementation allowing us to extend and modify action semantics. For example, a new facet seems necessary to provide elegant semantic descriptions of complex concurrent languages.

References

[1] A. Bondorf and J. Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[2] D. F. Brown, H. Moura, and D. A. Watt. Actress: An action semantics directed compiler generator. In *Proc. of CC'92, 4th International Conference on Compiler Construction*, volume 641 of *LNCS*, pages 95–109. Springer-Verlag, 1992.

[3] Y. Futamura. Partial evaluation of computation process – An approach to a Compile-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[4] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: a nonstrict, purely functional language, version 1.2. Technical Report YALEU/DCS/RR-777, Yale University Department of Computer Science, Mar. 1992.

[5] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[6] R. Milner. Theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17(3):348–375, 1978.

[7] P. D. Mosses. *Action Semantics.* Cambridge Press, 1992.

[8] P. D. Mosses. Foundations of modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, LNCS, 1999. to appear.

[9] P. D. Mosses. A modular SOS for Action Notation. Incomplete draft, Mar. 1999.

[10] H. Moura. An implementation of action semantics. Draft, 1993.

[11] J. Palsberg. A provably correct compiler generator. In *Proc. of ESOP'92 European Symposium on Programming*, volume 582 of *LNCS*, pages 418–434. Springer-Verlag, Feb. 1992.

[12] G. D. Plotkin. A Structural approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, Sept. 1981.

[13] P. L. V. Roy and A. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, Jan. 1992.

[14] D. Sahlin. An automatic partial evaluator for full Prolog. Ph.D. thesis TRITA-TCS-9101, Kungliga Tekniska Hgskolan, Stockholm, Sweden, 1991.

[15] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[16] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[17] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[18] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[19] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[20] P. Wadler. Linear types can change the world! In *Proc. of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, Apr. 1990.