Journal of Universal Computer Science, vol. 9, no. 8 (2003), 910-934
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

Object-Oriented Action Semantics Specifications

Claudio Carvilhe
(Catholic University of Paraná, Brazil
carvilhe@ppgia.pucpr.br)

Martin A. Musicante
(Federal University of Paraná, Brazil
mam@inf.ufpr.br)

Abstract: Action Semantics is a framework for the formal specification of programming languages. Two different, recently proposed approaches provide modularity to the framework, allowing for specification reusability and extension. In this work, we analyze the previous approaches, and introduce *Object-Oriented Action Semantics*, a new form of modular organization of Action Semantics descriptions. Object-oriented Action Semantics does not modify the syntax in which actions are written; the addition of object-oriented features (like classes and objects) is done as an upper layer to the semantic entities and functions. A simple Pascal-like, imperative programming language is described using the formalism. The extension and reuse capabilities of Object-Oriented Action Semantics are demonstrated by adding new features to the description. The semantics of the object-oriented action notation is also presented.

Key Words: formal semantics, action semantics, object-oriented specification.
Category: F.3.2, D.3.1, D.3.2

1 Introduction

Action Semantics [Mosses, 1992, Watt, 1991, Mosses, 1999] is a formal framework for the definition of programming languages. One of the goals of the Action Semantics project is to provide a notation that is both formal and suitable to be used in the description of real programming languages. Action Semantics descriptions have shown to have good reusability and extensibility properties [Mosses and Musicante, 1994]; however, the standard Action Semantics notation lacks of syntactic support for the definition of libraries, whose components would be reused in new descriptions [Labra Gayo, 2002].

Two solutions for this problem have been addressed by different authors. In [Doh and Mosses, 2003], the standard Action Notation is extended to define *modules*. In [Menezes and Moura, 2001] the notation is adapted to define *components*. These solutions have important pros and cons.

In this work, we propose the use of Object-Oriented concepts for the definition of a new extension of Action Notation. Our proposal tries to solve some problematic aspects of the previous Action Semantics extensions. A main advantage of our approach is the use of the standard Action Notation for the specifications “in-the-small”, combined with class constructors, to provide an object-oriented way of composing specifications.

This work is organized as follows: The next section briefly presents Action Semantics. Sections 3 and 4 summarize the introduction of modules and components in the formalism. Section 5 introduces Object-Oriented Action Semantics by means of an example. The operational semantics of our notation is also given in this section. Section 6 presents a case study: a simple imperative language is specified. This description is extended, in order to demonstrate the capabilities of our proposal. Section 7 is devoted to the conclusions of this work.

2 Action Semantics

Action Semantics [Mosses, 1992], [Watt, 1991], [Mosses, 1999] is a formal framework for describing programming languages semantics. In Action Semantics, the meaning of each phrase of a language is specified using *ad-hoc* entities called *actions*. Actions can be *performed*, processing data. Actions are defined using a special *Action Notation*, which defines *basic actions* and *action combinators*. Action semantics uses English words to enhance readability.

As in operational or denotation semantics, specifications in Action Semantics are traditionally structured using *semantic functions* and *semantic equations*.

In Action Semantics, the meaning of each phrase of a language is represented in terms of special entities called *actions*. Actions can be *performed* to process *information*, with various possible outcomes: normal termination (performance of the action *completes*), exceptional termination (it *escapes*), unsuccessful termination (it *fails*) or non-termination (it *diverges*). Action notation provides some primitive actions, and various *combinators* for forming complex actions, corresponding to the main fundamental concepts of programming languages.

A *data notation* is used to describe the information processed by actions. The standard data notation (included in action notation) provides a collection of algebraically defined abstract data types, including numbers, characters, strings, sets, tuples, maps, etc.; further data may be specified *ad hoc*.

There is also a third class of entities in action notation, called *yielders*. A yielder represents data whose value depends on the current information available to the primitive action in which it occurs. Yielders are *evaluated* to yield data. An example of a standard yielder is `the data bound to` I, which depends on the current bindings that are received by the enclosing primitive action.

Action notation possesses five so-called ‘facets’:

Basic: This facet deals with pure control flow, without reference to information processing issues. This facet includes combinators for sequencing, interleaving and non-deterministic, angelic choice between actions. For example, the compound action

A_1 `and then` A_2

performs the action A_1 first; The action A_2 is performed after A_1 completes.

Functional: This facet deals with *transient* data, which is given to or by an action. For example, when the primitive action

```
give the successor of the given natural
```

is given a natural number n as transient data, it completes, giving $n + 1$ as a transient. The compound action

A_1 `then` A_2

performs the action A_1 first; all transient data given by A_1 is passed on to A_2, which is performed after A_1 completes.

The primitive action `choose` D, where D is a sort of data, makes a non-deterministic choice of an individual of sort D, giving the chosen datum as a transient.

Declarative: This facet deals with the manipulation of *scoped* information, represented by associations of *tokens* to bindable data. For example, performance of the primitive action

```
bind ``max-length'' to 256
```

completes, producing a binding of the token `` ``max-length'' `` to the natural number 256.

Imperative: This facet is concerned with *storage* handling. A storage in action notation is simply a mapping from (the currently allocated) *cells* to storable data. For example, consider the action

```
  allocate a cell
then
  store 26 in the given cell
```

This action combines features of the functional and imperative facets. Notice that indentation was used to indicate precedence of operators and combinators.

Communicative: This facet provides a system of *agents*, which can each be 'contracted' to perform particular actions. Initially only a special 'user' agent is active. Agents can communicate using asynchronous message passing: the sending of a message is non-blocking. Each agent has its own *communication buffer*, in which all the messages sent to the agent are placed. Communication is reliable, in the sense that no message can be lost during transmission; however, there is no bound to the amount of time taken for a message to reach its destination agent. Moreover, each agent is created with its own storage, which cannot be affected (nor inspected) by other agents. Arbitrary data can be contained in messages, including the identities of agents.

Most of the primitive actions have a use in connection with only one facet each, but the action combinators generally involve a mixture of the basic, functional, and declarative facets, determining the flow of control, transient data, and bindings between the subactions.

Encapsulation of actions as data is also provided within action notation. This feature gives a simple way to support the description of procedure and function abstractions in programming languages. An *abstraction* is an item of data which encapsulates an action. Abstractions can be *enacted*; this operation results in the performance of the encapsulated action. Both transients and bindings can be supplied to abstractions before their enaction, for use by the encapsulated action. Abstractions can be treated just like any other data, i.e., given as transients, bound to tokens, stored in cells, and sent in messages. They are also used to determine the 'contracts' offered to agents in the communicative facet.

Apart from the communicative facet (which deals with communicating actions), all of the facets and combinators described above are used for specifying "in the small". They define the internal organization of a processing block.

As reported in [Doh and Mosses, 2003], Action Semantics specifications are *inherently modular*. However, in standard Action Semantics there are no constructors available to explicitly define and compose reusable blocks. Such capability would constitute an important feature in the context of the creation of new languages specifications, based on existing ones.

Two methods were recently proposed to improve modularity. Both methods are summarized in the following sections.

3 Adding modular structure to Action Semantics

In [Doh and Mosses, 2003], the use of *modules* to structure Action Semantics descriptions is proposed. In this approach, the specification of a programming language is constructed by the combination (or extension) of modules. Each module is specified by two main sections: Syntax and Semantics. The first section defines the syntax of the phrases to be defined by the module. The semantics part defines the meaning of these phrases, using (action) semantic functions.

The following example (adapted from [Doh and Mosses, 2003]) illustrates the definition and extension of a module to define arithmetic expressions.

```
Base Expressions {
    syntax:
        Expr.
    semantics:
        datum >= expressible.
        (*) evaluate: Expr -> action [giving an expressible]
}
```

The module above states that `Expr` is a syntactic sort, and that `evaluate` is a semantic function, taking an expression and being defined by an action. This module captures the basic behavior of expressions.

The module can be extended to define any kind of expressions. For example, the definition of arithmetic expressions can be done by defining a new module (which *imports* the above one), as follows:

```
Arithmetic Expressions {
    import Base Expressions.
syntax:
    Expr ::= Num | [[ Expr "+" Expr ]].
    Num ::= [[ digit+ ]].
    E1, E2 : Expr; N: Num.
semantics:
    expressible >= natural.
    (1) evaluate N = give decimal string-of-characters N.
    (2) evaluate [[ E1 "+" E2 ]] =
        ( evaluate E1 and evaluate E2) then
            give the sum(the given natural #1,
                         the given natural#2).
}
```

The module `Arithmetic Expressions` defines the syntax and semantics of sum expressions (of natural numbers). The semantics of these expressions is given in the usual Action Semantics style.

Other specific expressions modules can be constructed, in order to define the syntax and semantics of different kinds of expressions, like Boolean Expressions, Relational Expressions, etc. All these modules can be imported into an `Expressions` module, to describe all the expressions of a given programming language.

Further composition of modules could lead us to the definition of a complete programming language. For instance, an imperative programming language can be described as a combination of library modules, as follows:

```
Imp Imperative Language {
    import Expressions, Commands, Declarations
syntax:
    Program ::= [[Decl ";" Cmd]].
    D:Decl; C:Cmd.
```

```
semantics:
    bindable >= cell.
    storable >= expressible.
    run [[D ; C]] = elaborate [[ D ]]
                    hence execute [[ C ]].
}
```

The organization of actions proposed by the modular Action Semantics facilitates the construction of module libraries, from which predefined modules can be used in new programming language projects.

A main advantage of the modular Action Semantics is that the actions used in the semantic functions can be exactly the same as in traditional Action Semantics; furthermore, the notation can be straightforwardly changed in accordance to the advent of new versions of Action Semantics, as long as its algebraic base does not change [Doh and Mosses, 2003]. This contributes to the easy transformation of previous specifications, at the same time in which only a small amount of retraining would be needed for users who already work with the framework.

It must be noticed that the module import operation has the effect of *merging* the involved modules, yielding to a *flat* organization of the items being defined. As noticed in [Doh and Mosses, 2003], this form of module combination can be problematic, since it can lead to inconsistencies, in some cases, when the same name is used to describe different language constructors.

Let us now resume the characteristics of the modular approach:

- The notation for defining modules is simple, improving the readability of the resulting specifications;
- Reusability is enhanced. Specification parts can be used in new specifications;
- Modules can be combined, including already existing ones in a current project;
- Modules combination is problematic, and can cause inconsistencies.

4 Component-based Action Semantics

Component-based Action Semantics [Menezes and Moura, 2001], allows the creation of generic components and component libraries, using some ideas present in Montages [Kutter and Pierantonio, 1997]. The specification of a programming language is constructed by the combination of building blocks, called *components*. As in the modular approach, components can contain both syntactic and semantic definitions.

Let us exemplify the use of components in the definition of a programming language. Our description is adapted from [Menezes and Moura, 2001]. Initially, an auxiliary generic component called `binary` is defined, to be used to define binary operators:

```
binary _ _ _ :: component, string, yielder -> component

binary c s y =
    syntax
        [[ c s c ]]
    semantics
    |   |   semantics of the c defined by the subtree #1
    |   and semantics of the c defined by the subtree #3
    then give y
```

The (component) operation `semantics of` takes a component and returns an action. In this case, to evaluate the sub-expressions of a sum.

The (higher order) component `binary` expects three parameters: a component (to define which kind of expressions are to be evaluated), a string, representing an operator (name) and a yielder, to represent the operation on values.

The above component is meant to be the *skeleton* of other components. It can be used to define a new component for arithmetic expressions:

```
Expression =  ... ++ binary Expression "+" (the sum of them)
                  ++ binary Expression "-" (difference of them)
                  ++ ...
```

The operator "`++`" is provided by the new notation to be the composition of components. The application of this operator permits to define the `Expression` component above as the combination of the components defining each constant and operation of the language.

Notice that in the final component, the semantics of each syntactic part is defined locally, having no need to merge and unify the sorts and operations that are being defined.

In order to define programming languages, the notation needs to include primitives to deal with the definition and extension of predefined modules. This is done in Component-based Action Semantics by the introduction of new syntactic constructors, to deal with ***languages***. Two primitives for language construction are the combinators `extends` and `and`. The former is used to build and extend languages, while the latter is used to combine languages to obtain new ones. The signature of these operations can be given as follows:

```
extends [ _ ] with _ :: identifier, component
                                     -> language description.

_ and _ :: language description, language description
                                     -> language description.
```

Let us exemplify the use of these combinators to construct the specification of an imperative programming language. Suppose that there already exist language descriptions for the declarations, expressions and commands of this language. In this case, the description of programs can be given by:

```
Imperative Language =
    |  Declarations and Commands and Expressions
    and
    |  extends [ Program ] with
    |  syntax [[ Declaration ";" Command ]]
    |  semantics
    |  |  |     semantics of the subcomponent #1
    |  |  hence semantics of the subcomponent #2
```

Let us now resume the characteristics of the component-based approach:

- Component notation and Programming Language Notation provide an advanced way to create components and to define language specifications;
- Components are generic and can be used in new projects, avoiding repetition;
- A component library can be defined;
- Functions and operators provided by the framework are cumbersome. Specifications can be obscure.

5 Object-Oriented Action Semantics

In the previous sections we present two approaches to the introduction of modularity in Action Semantics. The first approach (Modular Action Semantics) has the advantage of using previous versions of Action Notation, without introducing new constructors for actions (new syntax is added just to define modules). However, the modular approach can be problematic in certain cases, since the locality of definitions is not enforced by the formalism. On the other hand, the component-based approach does not have this problem, but at the cost of significantly changing the syntax of actions, as well as its usage.

In both approaches, specification parts can be reused in new projects. A module or component can be written in an abstract way and extended to meet actual needs. The readability is increased in relation to traditional Action Semantics, since small specification parts can compose a large project.

In this context, we introduce (yet) another approach for the organization of specifications in Action Semantics. Our proposal, consists in the use of Object-Oriented concepts for structuring specifications.

An Object-Oriented Action Semantics specification consists on the definition of a ***hierarchy*** of objects and classes, in which the methods and attributes are defined by means of a standard Action Semantics semantic functions and data. The class hierarchy for a given language is derived from its syntactic structure. Each language phrase (declarations, commands, expressions, etc) is represented by one or more *objects*.

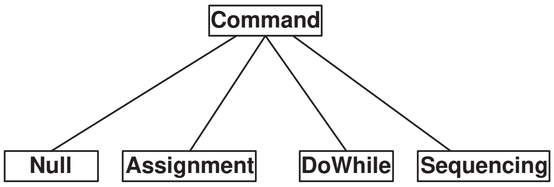

Figure 1: Toy command language class hierarchy.

Let us exemplify these concepts using a toy language of commands, whose syntax is defined as follows:

$$
\begin{aligned}
Cmd ::= \;& \text{“}\texttt{skip}\text{”} \\
\mid \;& Id := Expr \\
\mid \;& \text{“}\texttt{do}\text{”}\ Cmd\ \text{“}\texttt{while}\text{”}\ Expr \\
\mid \;& Cmd\ ;\ Cmd
\end{aligned}
$$

The non-terminal symbol *Cmd* defines the commands present in the language: the null command, assignments, iterations and sequences. There exist some features that are common to all commands. However, each one of them presents a particular structure and meaning. Such context can be represented using the *class hierarchy*, defined in Figure 1. In that figure, each box represents a class. The class Command should contain all the features common to all commands. Specific features can be expressed by the Command sub-classes: Null, Assignment, DoWhile and Sequencing.

As in the previous sections, we shall organize classes in Object-Oriented Action Semantics as having two basic parts: syntax and semantics. The top level class of commands can be represented as follows.

```
Class Command
        syntax:
            Cmd
        semantics:
            execute[[ _ ]] : Cmd -> Action
End Class
```

The class `Command`, detailed above, is defined to be the basic class for commands. In the syntax section (`syntax`), the syntactic sort `Cmd` is introduced. This sort will be extended by the subclasses of `command` to contain all the command syntactic trees.

A semantic function '`execute [[ _ ]]`' is defined in the semantics section (`semantics`) indicating a mapping from syntactic sort `Cmd` to an `Action`. It is worth noticing that the sort `Action` corresponds to the same sort as in traditional Action Semantics.

In our framework, semantic functions and semantic equations are respectively similar to methods and method definitions [Rumbaugh, 1994], [Meyer, 1997]. The semantic function '`execute`' is seen as a method of the class `Command`, and can be overloaded by its sub-classes. (This concept is usually called polymorphism by the Object-Oriented community.)

Let us now construct the sub-class of `command` dealing with the do-while loop:

```
Class Do-WhileCommand
        extending Command
        using E:Expression, C:Command
        syntax:
            Cmd ::= "do" C "while" E
        semantics:
            execute[[ "do" C "while" E ]]=
               unfolding
               ( (execute C and then evaluate E)
                 then
                 (unfold else complete))
End Class
```

The `extending` directive indicates that the `Do-WhileCommand` class is a sub-class of `Command`. Two objects (`E:Expression` and `C:Command`) are instantiated by the `using` directive. These objects will match the sub-phrases of commands.

The syntax section of the class adds a new command structure, redefining the syntax tree (`Cmd`). The '`execute`' method is then overloaded and the do-while command Action Semantics is given.

The classes `Command` and `Expression` must provide the methods '`execute`' and '`evaluate`', respectively. These methods will be called within the definition of the method `execute=` for the `Do-WhileCommand` class.

The next subsection extends standard Action Notation to express object-oriented concepts. The big-step operational semantics of the notation is given in section 5.2.

5.1 Syntax

Specifications in Object-Oriented Action Semantics are structured as a finite set of classes. The relationship between classes is specified using directives, indicating which objects will be instantiated by the actual class, as well as its position in the class hierarchy. The syntactic structure for classes is defined next. This syntax definition simply extends the standard Action Notation to include the class apparatus.

(1) *Class-Module* ::= “Class” *Class-Name* *Class-Body* “End-Class”

(2) *Class-Body* ::= ⟨ “extending” *Base-Class-Name* ⟩?
⟨ “using” *Objects-Declaration* ⟩? ⟨ *Class-Definition* ⟩?

(3) *Base-Class-Name* ::= *Class-Name*
| *Class-Name* “::” *Base-Class-Name*

(4) *Objects-Declaration* ::= *Object-Declaration*
| *Object-Declaration* “,” *Objects-Declaration*

(5) *Object-Declaration* ::= *Identifier* “:” *Class-Name*

(6) *Class-Definition* ::= “syntax” “:” *Syntactic-Part*
“semantics” “:” *Semantic-Part*

(7) *Syntactic-Part* ::= *TokenName* | *TokenName* “::=” *syntax-tree*

(8) *Semantic-Part* ::= ⟨ *Semantic-Functions* ⟩? ⟨ *Semantic-Equations* ⟩?

(9) *Semantic-Functions* ::= *Semantic-Function*
| *Semantic-Function* *Semantic-Functions*

(10) *Semantic-Function* ::= *Function-Name* “[[” “_”* “]]” *Tokens*
“->” ‘ ‘*Action*”
| *data*

(11) *Tokens* ::= *TokenName* | *TokenName* “,” *Tokens*

(12) *Semantic-Equations* ::= *Semantic-Equation*
| *Semantic-Equation* *Semantic-Equations*

(13) *Semantic-Equation* ::= *Function-Name* “[[” *syntax-tree* “]]”
‘ ‘=” *Action*

(14) *Action* ::= “complete” | “fail” | “unfold”
| *Action* “and” *Action* | *Action* “and then” *Action*
| “unfolding” *Action* | “give” *Yielder*
| “check” *Yielder* | *Action* “then” *Action*
| “bind” *Yielder* “to” *Yielder* | “furthermore” *Action*
| *Action* “hence” *Action* | *Action* “moreover” *Action*
| *Action* “before” *Action* | “store” *Yielder* “in” *Yielder*
| “deallocate” *Yielder* | “allocate a” *Sort*
| “enact” *Yielder* | *Action* “else” *Action*
| “recursively” “bind” *Yielder* “to” *Yielder*

(15) *Yielder* `::=` "`the`" *Sort* "`#`"*n* **|** "`the`" *Sort* "`bound`" "`to`" *k*
| "`the`" *Sort* "`stored`" "`in`" *Yielder*
| *Yielder* "`with`" *Yielder* **|** "`closure`" *Yielder*
| "`abstraction of`" *Action*

(16) *Sort* `::=` "`bindable`" **|** "`cell`" **|** "`cell`" "`[`" *Sort* "`]`"
| "`storable`" **|** "`abstraction`" **|** "`datum`" **|** "`integer`"
| "`value`" **|** "`truth-value`" **|** `Sort` "`|`" `Sort`

Rules (1) to (13) define the structure of classes in our approach, as exemplified in section 5. Rules (14) to (16) incorporate Action Semantics to classes.

The following subsection defines the operational semantics of this notation.

5.2 Semantics

Base-classes defined using our notation are *implicitly* part of a hierarchy. We shall define a class to which all classes belong. This *super-class* is called *State*. All classes defined in Object Oriented Action Semantics are sub-classes of *State*.

The class *State* has generic attributes, corresponding to transient information, bindings and storage; and provide *operations*, allowing us to handle such attributes. These operations are all the basic actions and action combinators. Both attributes and operations of *State* are visible to its sub classes.

Notice that although this is a new way of looking at actions, nothing is changed in the practical use of them to define semantic functions.

We define the *State* methods behavior by means of a big-step operational semantics [Winskel, 1993], characterized by the following relation:

$$B, t, b, s \vdash o \triangleright o', t', b', s'$$

The relation schema above identifies B as a *methods environment* (containing references to all user-defined methods), t as the transient information, b as bindings and s as the current store. We say that the operation (action) o, when performed, produces the outcome o', together with transient information t', bindings b' and storage s'. The definition of this relation is adapted from [Moura, 1993].

Let us now give some examples of the definition of the big step operational semantics for actions in our context.

$$B, t, b, s \ \vdash \ \texttt{complete} \ \triangleright \ \texttt{completed,} \ \{\}, \{\}, s \tag{1}$$

$$B, t, b, s \ \vdash \ \texttt{fail} \ \triangleright \ \texttt{failed,} \ \{\}, \{\}, s \tag{2}$$

$$\frac{\begin{array}{c} B,t,b,s \ \vdash \ a_1 \ \rhd \ \texttt{completed,} \ t_1,b_1,s_1 \\ B,t,b,s_1 \ \vdash \ a_2 \ \rhd \ \texttt{completed,} \ t_2,b_2,s_2 \\ \texttt{mergeable} \ t_1 t_2 \qquad \texttt{mergeable} \ b_1 b_2 \end{array}}{B,t,b,s \ \vdash \ a_1 \ \texttt{and then} \ a_2 \ \rhd \ \texttt{completed,} \ t_1 \oplus t_2, b_1 \oplus b_2, s_2} \tag{3}$$

$$\frac{B,t,b,s \ \vdash \ a_1 \ \rhd \ o_1,t_1,b_1,s_1 \qquad o_1 \neq \texttt{completed}}{B,t,b,s \ \vdash \ a_1 \ \texttt{and then} \ a_2 \ \rhd \ o_1,t_1,b_1,s_1} \tag{4}$$

$$\frac{\begin{array}{c} B,t,b,s \ \vdash \ a_1 \ \rhd \ \texttt{completed,} \ t_1,b_1,s_1 \\ B,t,b,s_1 \ \vdash \ a_2 \ \rhd \ o_2,t_2,b_2,s_2 \\ o_2 \neq \texttt{completed} \end{array}}{B,t,b,s \ \vdash \ a_1 \ \texttt{and then} \ a_2 \ \rhd \ o_2,t_2,b_2,s_2} \tag{5}$$

Rules (1) and (2) define the semantics of the operations `complete` and `fail`, respectively. Both of them terminate. The former represents successful termination, while the latter finishes without success. Rules (3), (4) and (5) define the behavior of the `and then` operation. Rule (3), establishes the behavior of the `and then` operation when both sub-operations (sub-actions) a_1 and a_2 complete. Operator ($\oplus$) defines *merging* of transient and bindings as defined in [Moura, 1993]. All the information received by the compound action is passed to each sub-action. If both sub-actions terminate with success, then the transients and bindings that resulted from the execution of these sub-actions are merged to compound the transients and bindings returned by the compound action. Rule (4) defines the behavior of the `and then` operation when the first sub-operation a_1 does not complete, and rule (5) establishes the behavior when sub-operation a_2 does not complete. In both cases, the result of the compound action is obtained from the unsuccessful sub-action, propagating its outcome.

The semantics of the other actions and yielders are given using the same specification style. The complete set of equations can be found in [Carvilhe, 2002].

5.2.1 Methods environment

All user-defined methods are stored in the methods environment. Two operations are needed to handle methods: insertion and fetch. The definition of these operations is given next.

Methods are syntactically structured as $f[\![H]\!] = O$, where H is a *syntax-tree schema* and O is an operation (action). A syntax tree schema is defined as a syntax tree with (free) variables standing for sub-trees.

The behavior of methods definitions is then defined by the following rules:

$$\frac{}{B \ \vdash \ f[\![H]\!] = O \ \rhd \ B[f[\![H]\!] \rightarrow O]} \tag{6}$$

$$\frac{(f[\![H]\!] \to O) \in dom(B), \qquad B,t,b,s \ \vdash \ O[H/h] \ \rhd \ o,t',b',s'}{B,t,b,s \ \vdash \ f[\![h]\!] \ \rhd \ o,t',b',s'} \qquad (7)$$

Rule (6) indicates that methods whose syntax is '$f[\![H]\!] = O$' are added to the methods environment. The notation $B[f[\![H]\!] \to O]$ indicates that a new method is inserted. The resulting method environment is similar to the original one (B), with the addition of a new method (f), which will be applied to syntactic trees that match the tree schema H. Rule (7) states that methods defined as $f[\![h]\!]$ produce the result o, t', b', s', provided that the method f exists in the methods environment B, and that it can be executed, after substituting the *syntax-tree schema* H by the matching syntax-tree h on the action O.

6 A case study - The μ-Pascal Language

In this section we present the use of Object-Oriented Action Semantics to define a toy language similar to Pascal. μ-Pascal is a simple imperative programming language which contains commands and expressions. The language possesses a block-structure.

The Object-Oriented Action Semantics of μ-Pascal is defined in this work to exemplify the extensibility properties of our framework. In section 6.2, the μ-Pascal Object-Oriented Action Semantics will be extended to a language with procedures and functions. μ-Pascal syntax is defined as follows.

(1) *Program* `::=` *Declaration* "`;`" *Command*

(2) *Declaration* `::=` "`var`" *Identifier* "`:`" *Type* ⟨ "`=`" `Expression` ⟩?
  | "`const`" *Identifier* "`:`" *Type* "`=`" *Expression*

(3) *Type* `::=` "`boolean`" **|** "`integer`"

A Program is formed by a declaration followed by commands. Constants and Variables can be declared indicating their type.

(4) *Command* `::=` "`let`" *Declaration* "`in`" *Command*
  | *Identifier* "`:=`" *Expression*
  | "`if`" *Expression* "`then`" *Command* ⟨ "`else`" *Command* ⟩?
  | "`repeat`" *Command* "`until`" *Expression*
  | *Command* "`;`" *Command*

(5) *Expression* `::=` "`true`" **|** "`false`" **|** *Numeral* **|** *Identifier*
  | *Expression* ("`+`" **|** "`-`" **|** "`*`" **|** "`<`" **|** "`=`") *Expression*

(6) *Identifier* `::=` *Letter* ⟨ *Letter* | *Digit* ⟩`*`

(7) $Numeral$ `::=` $Digit$ $\langle Digit \rangle$`*`

Commands in μ-Pascal contain assignments, selections, iterations and sequences. Expressions can be arithmetical or logical.

6.1 μ-Pascal Semantics

The syntax of the language is our departing point to construct the class hierarchy of the specification. The syntactic rules of the language definition can be mapped into objects which incorporate syntax and semantics. Some diagrams will be constructed, using a tree notation to express the class hierarchy.

Let us begin by constructing classes to represent the language identifiers, numerals and types. After this phase, we define the declarations, commands and expressions hierarchy to express the meaning of the language.

```
Class Identifier
        syntax:
            Id ::= letter [ letter | digit ]*
End Class

Class Numeral
        syntax:
            N ::= digit+
        semantics:
            valuation [[ _ ]] : N -> integer
End Class

Class Type
        syntax:
            T ::= "boolean" |   "integer""
        semantics:
            allocate-for-type [[ _ ]] : T -> Action
            allocate-for-type[[ "boolean" ]] =
                        allocate a cell [ truth-value ]
            allocate-for-type[[ "integer" ]] =
                        allocate a cell [ integer ]
End Class
```

The class `Identifier` represents names (variables and constants). This class has only syntactic definition.

The class `Numeral` represents integer numbers. In the `semantics` section a `valuation [[ _ ]]` method is defined to map numerals to integers. The definition of the `valuation` function is straightforward. As it is usual in semantic descriptions, the double bracket notation is used to remark that the function parameter is a syntactic entity.

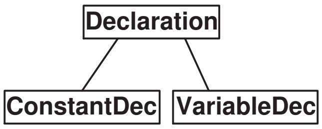

Figure 2: μ-Pascal declarations hierarchy.

The class `Type` represents the language data types. The method `allocate-for-type` is defined, establishing truth-value or integer data types allocation.

Constants and variables can be declared in μ-Pascal. The declarations hierarchy can be constructed in accordance to Figure 2, as follows.

```
Class Declaration
        using I:Identifier, E:Expression
        syntax:
            Dec
        semantics:
            elaborate [[ _ ]]: Dec -> Action
End Class
```

The class `Declaration` instantiates two objects: `I` and `E`, respectively of classes `Identifier` and `Expression`. Both objects are not used by the class itself. They are defined at this point to be available to the sub-classes of `Declaration`. Notice that the class hierarchy allows us to instantiate those objects only in the base-class `Declaration`, inhibiting repetition. A non-terminal symbol `Dec` is introduced at the syntax section to be redefined by the sub-classes. A method called `elaborate [[ _ ]]` is declared in the semantics section, to be overloaded by the sub-classes.

Let us now give a sub-class of `Declaration`, to define the meaning of constant definitions:

```
Class ConstantDec
        extending Declaration
        using T:Type
        syntax:
            Dec ::= "const" I ":" T "=" E
        semantics:
            elaborate[["const" I ":" T "=" E]]=
                evaluate E then
                bind I to the value
End Class
```

The `Dec` token is redefined in the `ConstantDec` class (above), establishing the syntax if constant definitions. In the semantics part, the `elaborate [[ _ ]]`

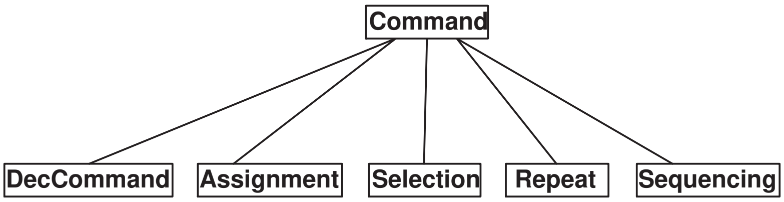

Figure 3: μ-Pascal commands hierarchy.

method is overloaded, to deal with the new syntax. The objects `I:Identifier` and `E:Expression` are used at this point, and the semantics of a constant declaration is defined using standard Action Notation.

The following class deals with variable declarations in μ-Pascal:

```
Class VariableDec
        extending Declaration
        using T:Type
        syntax:
            Dec ::= "var" I ":" T ["=" E]
        semantics:
            elaborate[["var" I ":" T]]=
                  allocate-for-type T
                  then bind token I to the cell #0
            elaborate[["var" I ":" T "=" E]]=
                    (evaluate E and allocate-for-type T)
                  then
                    (bind I to the cell #1
                     and store the value #0 in the cell #1)
End Class
```

The `Dec` token is redefined in the `VariableDec` class (above), establishing the structure for variable declarations. This class instantiates a `T:Type` object. The variable declaration syntax is redefined at the syntax section. The `elaborate [[ _ ]]` method is redefined twice, considering the absence (or not) of an expression `E`. The `T:Type` object is not defined in the base-class `Declaration`, but it is used both in `ConstantDec` and `VariableDec`. The semantics of a variable declaration is defined using standard Action Notation.

The commands class hierarchy is depicted in Figure 3. The main class of this hierarchy is `Command`, defined as follows.

```
Class Command
        syntax:
            Com
        semantics:
            execute [[ _ ]] : Com -> Action
End Class
```

In the syntax section, the token `Com` is introduced to represent the syntactic sub-trees of commands. The method `execute [[ _ ]]` is declared in the semantics section to be redefined by the sub-classes of `Command`. These sub-classes are defined next.

```
Class DecCommand
        extending Command
        using D:Declaration, C:Command
        syntax:
            Com ::= "let" D "in" C
        semantics:
            execute[["let" D "in" C]]=
                (furthermore elaborate D)
                hence execute C
End Class
```

The class `DecCommand` defines the syntax and semantics of command blocks. Two objects are instantiated by `DecCommand` class: `D:Declaration` and `C:Command`. In the syntax section the sort `Com` is redefined to express command blocks. In the semantics section the (standard) Action Semantics description of blocks is given.

```
Class Assignment
        extending Command
        using I:Identifier, E:Expression
        syntax:
            Com ::= I ":=" E
        semantics:
            execute[[I ":=" E]]=
                  evaluate E
                then
                  store the value in the cell bound to I
End Class
```

The class `Assignment`, above, instantiates two objects `I:Identifier` and `E:Expression`. In the syntax section the structure of assignments is defined. The semantics of an assignment is given by the evaluation of the expression, followed by the storage of the resulting value. The remaining classes of the command hierarchy (`Selection`, `Repeat` and `Sequencing`) are structured in the same way as `DecCommand` and `Assignment`; they are omitted here.

Let us now define the expressions of the language, whose class hierarchy is defined in figure 4.

The super-class `Expression`, defined below, presents a similar structure to that of `Command`. In the syntax section a token `Exp` is introduced to repre-

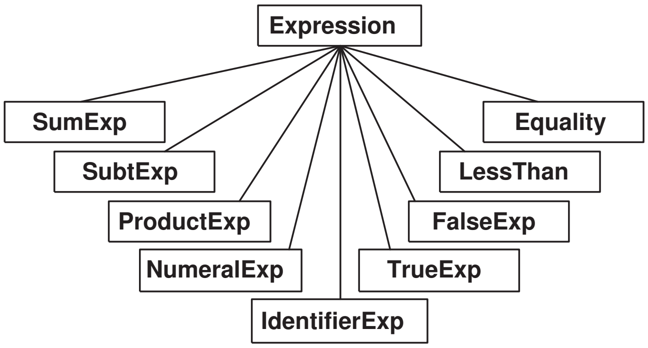

Figure 4: μ-Pascal expressions hierarchy.

sent the syntactic sub-trees of expression. In the semantics section the method `evaluate [[ _ ]]` is introduced.

```
Class Expression
        syntax:
            Exp
        semantics:
            evaluate [[ _ ]] : Exp -> Action
End Class
```

All the subclasses of `Expression` can be defined in a similar way as the subclasses of `Command`.

The definition of the addition operation can be given as:

```
Class SumExp
        extending Expression
        using E1:Expression, E2:Expression
        syntax:
            Exp ::= E1 "+" E2
        semantics:
            evaluate[[ E1 "+" E2 ]] =
                (evaluate E1 and evaluate E2)
                then give sum(the integer#1, the integer#2)
End Class
```

The class `SumExp`, instantiates two objects from classes `Expression` (`E1` and `E2`). The syntactic sort `Exp` is redefined in the syntax section. The semantics of a sum expression is specified by evaluating both sub-expressions and adding their results. All the operators of the language can be defined in a similar manner.

Let us now define the meaning of identifiers within expressions:

```
Class IdentifierExp
    extending Expression
    using I:Identifier
    syntax:
        Exp :: = I
    semantics:
        evaluate [[ I ]] =
            (give the value bound to I) or
            (give the value stored in the cell bound to I)
End Class
```

The class `IdentifierExp` instantiates an object `I:Identifier`. The syntax of expressions is extended to represent identifiers. In the semantics section, the meaning of a identifier is defined. Notice that an identifier can be a constant or variable; both cases are covered by the previous specification by using the action combinator '`_ or _`'.

The other classes of expressions can be defined using the same principles.

Once the language's declarations, commands and expressions are represented using hierarchies, it is possibly to specify a class representing the whole μ-Pascal language:

```
Class muPascalLanguage
        using D:Declaration, C:Command
        syntax:
            Prog ::= D ";" C
        semantics:
            run[[ _ ]] : Prog -> Action
            run[[ D ";" C ]] =
              elaborate D hence execute C
End Class
```

Two objects are instantiated in the `muPascalLanguage` class: `D:Declaration` and `C:Command`. The `Prog` token defines the μ-Pascal program general structure. In the semantics section, a `run[[ _ ]]` method is defined mapping a program to an action.

6.2 Extending the Specification - The m-Pascal Language

Let us now use the class hierarchy defined in the previous sections, to create a new language, which we will call m-Pascal. The language incorporates procedure and function abstractions to μ-Pascal. Let us now extend the language to contain the new structures:

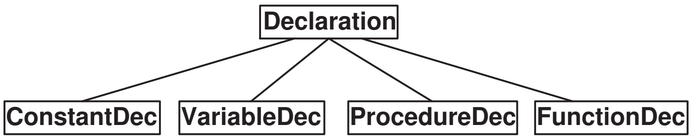

Figure 5: m-Pascal declarations hierarchy.

(1) *Declaration* `::=` ...
  | “`proc`” *Identifier* “`(`” *Formal-Par* “`)`” “`=`” *Command*
  | “`func`” *Identifier* “`(`” *Formal-Par* “`)`” “`=`” *Expression*

(2) *Formal-Par* `::=` “`var`” *Identifier* “`:`” *Type*

(3) *Command* `::=` ... | “`call`” *Identifier* “`(`” ⟨ *Actual-Par* ⟩ “`)`”

(4) *Expression* `::=` ... | *Identifier* “`(`” ⟨ *Actual-Par* ⟩ “`)`”

(5) *Actual-Par* `::=` *Expression*

The abstract syntax of declarations is extended by rule (1) to define procedure and function definitions. Commands and expressions are also changed in (3) and (4) respectively, to include procedure and function calls.

The objects containing the definitions of procedures and functions are expressed by extending the `Declaration` class defined in the μ-Pascal specification. The declarations hierarchy will be changed to include two new sub-classes: `ProcedureDec` and `FunctionDec` (Figure 5):

```
Class ProcedureDec
    extending Declaration
    using C:Command, FP:FormalPar
    syntax:
        Dec ::= "proc" I "(" FP ")" ":" C
    semantics:
    elaborate[["proc" I "(" FP ")" ":" C]] =
        recursively bind I to
          (closure abstraction of
                     (furthermore elaborateFP FP
                      hence execute C))
End Class
```

The class `ProcedureDec`, gives the syntax and semantics of procedure declarations. The class representing function declarations is similar to the above one:

```
Class FunctionDec
    extending Declaration
```

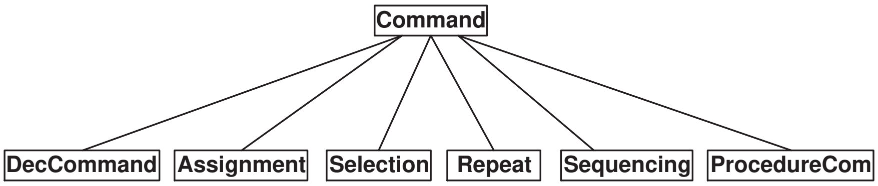

Figure 6: m-Pascal commands hierarchy.

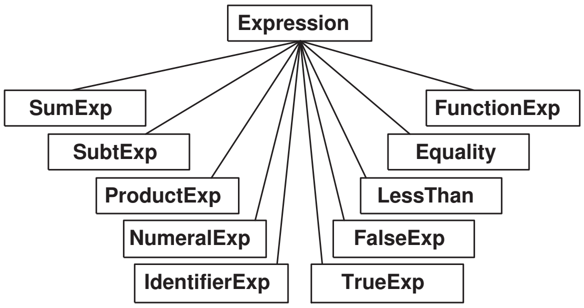

Figure 7: m-Pascal expressions hierarchy.

```
    using FP:FormalPar
    syntax:
        Dec ::= "func" I "(" FP ")" ":" E
    semantics:
    elaborate[["func" I "(" FP ")" ":" E]] =
        recursively bind I to
          (closure abstraction of
                   (  furthermore elaborateFP FP
                     hence
                      (evaluate E then give the value)
                   )
          )
End Class
```

Notice that an auxiliary class `FormalPar`, which represents formal parameters, is instantiated by `ProcedureDec` and `FunctionDec`; its definition is straightforward, and it is omitted here.

Procedure calls and function applications require additions to the `Command` and `Expression` hierarchies. These changes are represented in Figure 6 and Figure 7, respectively. The classes corresponding to the new language constructions are given below:

```
Class ProcedureCommand
    extending Command
    using I:Identifier, AP:ActualPars
    syntax:
       Com ::= "call" I "(" AP ")"
    semantics:
       execute[["call" I "(" AP ")"]] =
           evaluateAP AP
         then
           enact (the procedure bound to I with the value)
End Class
```

```
Class FunctionExpression
    extending Expression
    using I:Identifier, AP:ActualPars
    syntax:
       Exp ::= I "(" AP ")"
    semantics:
       execute[["call" I "(" AP ")"]] =
           evaluateAP AP
         then
           enact(the function bound to I with the value)
End Class
```

The class definitions above use an auxiliary class `ActualPar`, which represents actual parameters and the operations over them. The definition of this class is omitted here for space reasons.

Object-Oriented features were applied to extend the μ-Pascal language. `ProcedureDec` and `FunctionDec` objects were created extending `Declaration`, to represent procedures and functions. Procedure calls were incorporated to `Command`, by the creation of the `ProcedureCommand` object and function applications were incorporated to expressions. This allows us to define the main class of the language specification as:

```
Class mPascalLanguage
        using D:Declaration, C:Command
        syntax:
            Prog ::= "declare" D "used-in" C
        semantics:
            run _ : Prog -> Action
            run[[ "declare" D "used-in" C ]] =
                elaborate D
                hence execute C
End Class
```

Notice that since the language was redefined at the declarations, commands and expressions levels, the body of the most general class for the language remains unchanged.

7 Conclusions

Object-Oriented Action Semantics is a new framework to organize Action Semantics descriptions. In this new framework, semantic functions are seen as *methods*, describing the meaning of constructions, defined locally within a class.

The hierarchical structure of the specifications makes it possible to choose which methods are to be applied to each syntactic object. Our proposal does not change the syntax of actions, maintaining the traditional style of writing Action Semantics specifications.

Reusability and extension are the main motivations of our proposal. Reusability is achieved in Object-Oriented Action Semantics by using class instantiation. Extension is improved by the definition of class hierarchies.

We have shown, in a simple case study, a way to construct objects, as well as their instantiation and specialization. In the case study, we have built a class hierarchy for a simple Pascal-like language. The specification of this language was then changed to add new syntactic constructions.

The (big step) operational semantics of the new notation is given. The only addition to the existent operational semantics of Action Notation is the use of a *methods environment*, to contain the definition of methods.

The characteristics of our proposal can be summarized as:

- Object-Oriented Action Notation is simple. The class structure is based on the *modules* notation introduced by [Doh and Mosses, 2003], and was inspired by the notions of classes and hierarchy from object-oriented programming;

- Reusability is improved. Instantiation and extension permit the construction of libraries of programming languages concepts, that can be instantiated in programming languages projects.

References

[Carvilhe, 2002] Carvilhe, C. (2002). Object-oriented action semantics. Master's thesis, Federal University of Paraná. Available at `http://www.ppgia.pucpr.br/~carvilhe/full`. (In portuguese).

[Doh and Mosses, 2003] Doh, K.-G. and Mosses, P. D. (2003). Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36. Elsevier Science Publishers.

[Kutter and Pierantonio, 1997] Kutter, P. and Pierantonio, A. (1997). Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442.

[Labra Gayo, 2002] Labra Gayo, J. (2002). Reusable semantic specifications of programming languages. In *SBLP 2002 - VI Brazilian Symposium on Programming Languages*, Rio de Janeiro, Brazil. Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio.

[Menezes and Moura, 2001] Menezes, L. C. and Moura, H. (2001). Component-based action semantics: A new approach for programming language specifications. In *SBLP 2001 - V Brazilian Symposium on Programming Languages*, pages 152–163, Curitiba, Brazil. Universidade Federal do Paraná.

[Meyer, 1997] Meyer, B. (1997). Object-oriented software construction. In *Object-Oriented Software Construction 2nd ed.* Prentice-Hall.

[Mosses, 1992] Mosses, P. D. (1992). Action semantics. In *Action Semantics*. Cambridge University Press.

[Mosses, 1999] Mosses, P. D. (1999). A modular SOS for Action Notation. Technical Report BRICS RS-99-56, University of Aarhus, Dep. of Computer Science.

[Mosses and Musicante, 1994] Mosses, P. D. and Musicante, M. A. (1994). An Action Semantics for ML concurrency primitives. Number 873 in Lecture Notes in Computer Science, Barcelona, Spain. FME, Springer-Verlag.

[Moura, 1993] Moura, H. (1993). In *Action Notation Transformations - Ph.D. Thesis*. University of Glasgow.

[Rumbaugh, 1994] Rumbaugh, J. (1994). In *Object-Oriented Modeling and design*. Campus.

[Watt, 1991] Watt, D. A. (1991). *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press.