A Framework for Defining Object-oriented Languages Using Action Semantics

L.C.S. Menezes, S.C.B. Soares, J.B. Meneses, H. Moura and A.L.C. Cavalcanti

Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil

{lcsm, scbs, jbm, hermano, alcc}@di.ufpe.br

Abstract

The benefits of formally specifying programming languages are widely recognized. Formalization is even more important for object-oriented languages whose design involves subtle interactions between several different concepts. Many formalisms have been proposed and Action Semantics is among the most successful. This paper defines a set of semantic entities useful for formally specifying object-oriented programming languages using Action Semantics. As a case study, we use these entities to describe the object-oriented kernel of Java.

Key words: semantic models, Java, action semantic descriptions.

1 Introduction

It is very important to define programming languages formally, providing an abstraction that helps with the design of the language, and a mathematical notation that allows a formal proof that a particular implementation of the language satisfies the specification. Many formalisms to specify programming languages have been proposed; typically each of them emphasizes and facilitates different aspects and applications of a programming language definition.

The use of an abstract and mathematical notation for the description of programming languages, which was the main source of inspiration in the design of Denotational Semantics, turned out to be inconvenient for the specification of real and full-scale programming languages. Actions Semantics cover some very desirable pragmatic properties that Denotational Semantics lacks, as readability, modularity, abstraction, comparability, and reasonability.

In [1,12,4,15,6], operational approaches are used to specify subsets of Java [5]. The language descriptions presented in [17,14,7] use Action Semantics. In [8] Action Semantics is also used to specify a subset of C++.

These Action Semantics descriptions independently define semantic entities and operators to support object-oriented concepts. The use of different semantic entities and operators by each language designer could be avoided if a library of object-oriented semantic entities and operators existed. Our aim is to define that library and, consequently, decrease the design time of object-oriented programming languages and improve the readability of specifications.

We consider many important concepts of object-oriented programming languages, like instance variables and methods, static variables and static methods, classes with single and multiple inheritance, dynamic binding, overwriting of methods, instance creation, null pointers, variables and methods visibility, shadowing of variables (variable hiding), and treatment of exception. The latter is actually already considered in Action Semantics.

In Section 2 we briefly describe Action Semantics, presenting examples of its notation. Then, in Section 3, we define the object-oriented semantic entities. After that we present a case study where we specify a subset of the Java programming language. In the sequence, in Section 5, we present a view of related works, and finally, in Section 6, the conclusion and future works.

2 Action Semantics

Action Semantics has all the desirable properties that formalisms for specifying programming languages should have [18]:

- *Readability.* The notation used is verbose and suggestive, which improves readability of semantic descriptions. The correspondence to well-known concepts found in programming languages also contributes to readability.
- *Modularity.* The semantics of a programming language is given in terms of a small number of standard primitive actions and action combinators. This provides the possibility of reusing parts of previous descriptions when specifying new languages. The *polymorphic* behavior of the action operators (regarding the different kind of information they process) allows their use to specify languages fundamentally different using the same set of operators. Moreover, language descriptions can be organized in modules.
- *Abstractness.* Action Semantics is operational in flavour, but not implementation-biased.
- *Comparability.* The use of standard primitive actions and action combinators also facilitates the semantic comparison of languages.
- *Reasonability.* The standard primitive actions and action combinators satisfy nice algebraic properties that can be used to reason about programs and allow us to carry out useful *transformations.*

Action Semantics describes the semantics of programming languages using an order sorted algebraic specification model named unified algebras [9] and a formal metalanguage named action notation.

Unified algebras define two kinds of entities: sorts and operators. Sorts are used to model abstract data types and represent a set of elements with some common properties. Operators take arguments of defined sorts and produce values of another specified sort. The terms of an algebraic specification are formed by the sorts defined by the specification and by the application of the defined operators over other terms.

Action notation contains structures that represent the main concepts found in existing programming languages. Action notation is formed by three kinds of entities: actions, yielders and data notation.

An action is an entity that can be performed. When an action is performed, it receives information from an outside environment, produces new information to the environment, and returns an outcome indicating the result of the action performance. Below we show some examples of action constructors.

- bind "x" to 10: this action produces the binding of the token "x" to the value 10.
- store true in cell1: this action stores the value true in the memory location cell1.

Yielders model expressions that can be evaluated during the performance of an action. The result of an yielder evaluation depends of the current environment passed to the current performing action. Below we give an example of one of the most important yielders defined by the action notation.

- the integer bound to "x": this yielder returns the integer value associated with the token "x" in the scoped information.

The data notation models data types used by actions and yielders. Numbers, strings, lists, and tuples are some examples of these datatypes. An example of a data notation for lists is presented below.

- head-of (_) :: list $\rightarrow$ datum: this operator returns the first element of a list.
- tail-of (_) :: list $\rightarrow$ list: this operator returns the given list without its first element.

An Action Semantics description for a programming language is an unified algebraic specification divided in the following modules:

- Abstract Syntax: describes the abstract syntax for the programming language.
- Semantic Functions: describes a mapping from the abstract syntax tree (AST) of programs to their meaning, using the action notation.
- Semantic Entities: defines the data types used by the language, and auxiliary sorts and operators used by the description in the previous module.

In the literature we can find many works using Action Semantics to specify programming languages such as Pascal [11], CCS and CSP [2], Standard ML [16] and Java [17]. For a detailed presentation of Action Semantics see [10].

3 Object-Oriented Semantic Entities

Specifying a complete object-oriented language using only the standard action notation is a difficult task. Such a specification requires complex semantic entities which are not given by action notation and, therefore, have to be provided by the designer. Specifications like this, which use "proprietary" notation, can become hard to read.

As a solution to this problem, we propose an extension of the action notation that includes object-oriented semantic entities. These new semantic entities are useful to describe languages like C++ and Java. This section describes these object-oriented semantic entities. A complete formal specification is too large to fit in this paper, but can be found in [3].

3.1 Tokens

The token defined in this object-oriented model is formed by a sequence of strings:

- token = string^+.

This definition differs from that usually found in common programming languages descriptions, which define tokens as single strings, because a single name may not be enough to identify a specific class entity. For instance, an object can have many overloaded versions of the same method, defined in different classes. In the present notation a method is identified using a pair containing the class name and the method name.

To make the specifications more readable, we define the data types class-name, method-name, and variable-name used to represent, respectively, class names, method names, and variable names in specifications. The first data type is defined as a single string and the others are defined as tokens. We also define the following special tokens: constructor, superclass and self. The sort constructor identifies the method that initializes a recently created object; the sort superclass represents the superclass of the class of the current object; finally, the sort self identifies the current object.

3.2 Objects

The object structure is initially left open to allow their extension by the designer:

- object = □ .

Therefore, we propose a standard simple object structure to represent an object. In this structure, an object is composed by a pair containing the class information and a set of bindings that stores the object internal structure. A binding is a mapping from tokens to bindable elements.

- object $\geq$ object of (class, bindings).

Some basic operators for objects are defined: the operator "t bound in o" is used to retrieve the meaning of the identifier t in the object o; the operator "o instance of c" tests if the object o is an instance of the defined class c; finally the operator "o cast to c" returns another object, produced from the object o, with static scope changed to fit in the scope requirements defined in class c.

3.3 Classes

The specification of the class structure is also left open:

- class = □ .

The object-oriented semantic entities define a basic class structure. A class is formed by a tuple whose elements are: a class name, a set of superclasses, a redirection, an object, and a set of class components.

- class $\geq$ class of (class-name, class*, redirects , object, class-component*).
- redirects = map [token to token].

The class name is a string used to identify the class. The set of superclasses contains the definitions of the superclasses. The redirection is used to identify class elements more precisely. It is defined as a function that maps single names (strings) to more complete names (with class name information). This structure is used to decide which version of an overridden class element should be used in some class scope. The object is used to store static variables and methods. Finally, the class components define the class body.

In this model we define a class as a subsort of object. This property helps to describe languages where classes can be seen as ordinary objects belonging to a special class. Java and Smalltalk are examples of this kind of language. This property is also useful to handle class (static) and instance (non-static) components with a uniform notation.

- class $\leq$ object.

The sort metaclass is defined to represent the class of classes when they are handled like objects:

- metaclass : class.
- metaclass = □ .

Its definition is language dependent and is left open.

3.4 Class Components

The object-oriented semantic entities specify three kinds of components which can be defined inside classes: constructors, methods, and variables.

- constructor _ :: method $\rightarrow$ class-component.
- _ method _ named _ :: qualifier, method, method-name $\rightarrow$ class-component.
- _ variable _ _ :: qualifier, type, variable-name $\rightarrow$ class-component.

The last two entities are qualified with flags indicating its type (static/non static) and access level (public, protected, private):

- qualifier = public | protected | private | static.

The constructor and method _ named operators define new methods in the class body. The constructor m operator defines a method m used to initialize a recently created instance of the class. The operator q method m named n defines a new ordinary method m named n. Both operators use the sort method to specify the method body. A method is just an alias to the Action Semantics sort abstraction. The definition of method considers that method is a subsort of the Action Semantics sort bindable. This is necessary to enable us to associate tokens with methods in bindings.

- method = abstraction.

  method $\leq$ bindable.

The design of the object-oriented semantic entities makes transparent to the designer the process of making binds to class components, simplifying the specification.

A variable definition contains information about its type and name. The type specification is shown below:

- type = □
- type $\geq$ class
- allocate for type _ :: type $\rightarrow$ action[allocating][giving a cell].

The type definition is left open because it is a language dependent feature. We define that classes are types and there is an operator (defined by the language specification) that allocates memory for types.

3.5 Handling Classes and Objects

The action that creates a class structure, returning it as a transient value, has the following signature:

- class named _ subclass of _ with _ ::

  class-name, class-name*, class-components $\rightarrow$ action[giving a class].

It is responsible to initialize the class fields and allocate memory to static variables, if any.

The action that creates a new instance of a given class is declared as follows:

- new instance of _ :: class $\rightarrow$ action [giving a object]

The recently created object is returned in transient information. If the instantiated class defines a constructor, it will be executed receiving the given data for the new instance of action.

The object-oriented semantic entities also define some auxiliary operators: the operator classes of c returns the set of superclasses of the class c; the operator c_1 is subclass of c_2 checks if the class c_1 is a subclass of the class c_2; finally, the operators create object c and create static object c are used to initialize non-static and static class components, respectively; the designer should extend them if new kinds of class components are created.

4 Case Study: Java Object Oriented Kernel

To illustrate the use of the semantic entities introduced in this paper, we describe a subset of the Java programming language that includes class declarations with inheritance, method calls, and object creation. Here we focus on the parts of the definition that make use of the object-oriented semantic entities just introduced. A complete description of this language is available in [3].

4.1 Abstract Syntax

A program in the specified subset of Java is formed by some class declarations followed by a block that defines the startup code for the program.

Program = ⟦ ClassDeclaration* "`in`" Block ⟧.

A class declaration is formed by the keyword "`class`" followed by the class name, the superclass declaration, and some declarations of class components between braces.

ClassDeclaration = ⟦ "`class`" Identifier SuperClass "{" ClassDecl* "}" ⟧.

SuperClass = ⟦ ⟨ "`extends`" Identifier⟩$^?$ ⟧.

There are three kinds of class component declarations in the language: variable declaration, abstract method declaration, and ordinary method declaration.

ClassDecl = ⟦ Qualifier Type Identifier ";" ⟧ **|**
⟦ "`abstract`" Qualifier Type Identifier "("
FormalArguments ")" ";" ⟧ **|**
⟦ Qualifier Type Identifier "(" FormalArguments ")" Block ⟧.

Qualifier = ⟦ ⟨ "`static`" **|** "`public`" **|** "`private`" **|** "`protected`" ⟩* ⟧.

A variable declaration is formed by a qualifier, a type, and a name. An abstract method declaration is formed by the keyword "`abstract`", a qualifier, a return type, a name, and an arguments declaration. An ordinary method declaration is formed by a qualifier, a return type, a method name, an arguments declaration, and a block of commands. The qualifier determines the visibility of the declared element and whether it is static or not.

When executing an object-oriented program, we handle the declared classes and objects using method calls and object-oriented expressions.

MethodCall = ⟦ Expression "." Identifier "(" Arguments ")" ⟧ **|**
⟦ "`super`" "." Identifier "(" Arguments ")" ⟧.

Method calls are commands beginning either by an expression (the object that receives the message), or the keyword "`super`", indicating that this method will be searched in the superclass of the current object; after that there is a method name and arguments.

```
OOExpression = [[ "new" Identifier "(" Arguments ")" ]] |
    MethodCall |
    [[ "super" "." Identifier ]] | [[ Expression "." Identifier ]].
```

Object-oriented expressions can be an instance creation, a method call, or a variable access.

4.2 Semantic Functions

This section defines the semantics of the object-oriented kernel of Java using the semantic entities described in this paper. The semantics of programs is defined by the following function.

- run _ :: Program → action.

```
run [[ d "in" b ]] =
    | pre-elaborate d before elaborate d
    hence
    | execute b.
```

To execute a program we need to prepare the environment defined by the program (operator pre-elaborate), build the environment declarations (operator elaborate), and finally execute the main program (operator execute).

- pre-elaborate _ :: Declaration* → action.

The semantic function pre-elaborate defines the global bindings used inside of the declared classes. Its definition uses only the standard action notation.

The semantic function elaborate declares classes defined by Java programs. To declare a class we must obtain the superclass defined by the declaration (operator retrieve superclass) and build the class components specified (operator build class components), create the class structure (operator class named), and finally produce the declaration using the primitive action rebind.

- elaborate_ :: ClassDeclaration → action.

```
elaborate [[ "class" i:Identifier s:SuperClass "{" d:ClassDecl "}" ]] =
    | | retrieve superclass s
    | and
    | | build class components d
    then
    | class named i subclass of (given classes # 1) with
    | (given class-components#2)
    then
    | rebind i to the given class.
```

Using the object-oriented notation proposed in this paper the designer does not need to have any knowledge of the internal structure used to represent classes. Also a more intuitive notation is available to make the semantics of class declarations simpler and the results more readable.

The semantic function build class components produces class components semantic entities from the syntactic entity ClassDecl. Here we show its signature and its definition for method declarations.

- build class components _ :: ClassDecl → action.

```
build class components ⟦ q:Qualifier Type i:Identifier "("
                           f:FormalArguments ")" b:Block ⟧ =
    | | get qualifier q
    | and
    | | give abstraction of
    | | | elaborate formal f before execute b
    then
    | give (given qualifier#1) method (given method#2) named i.
```

Initially we retrieve the method qualifier (semantic function get qualifier) and build the method abstraction (operator abstraction of). Finally we use the produced data to build the class component using the operator method _ named of the object-oriented semantic entities. The most complex tasks involved in declaring methods (overridding superclass methods, binding instance variables, and hiding private declarations, for instance), common in most object-oriented languages, are handled internally by the operator method _ named.

The semantic function get qualifier just builds class components from their equivalent syntactic entities. The semantic function execute specifies how to execute commands of Java. Its signature and definition for method calls is described below.

- execute _ :: MethodCall → action.

```
execute ⟦ e:Expression "." i:Identifier "(" a:Arguments ")" ⟧ =
    | | evaluate e
    | and
    | | evaluate argument a
    then
    | enact application of
    | | the method i bound in (given object#1) to (rest of given data).
```

Initially the expression indicating the object that receives the message and the arguments are evaluated; afterwards the action retrieves the abstraction related with this method and object (operator bound in) and executes the method (action enact). This is another example of how the object-oriented semantic entities can simplify the description of object-oriented programming languages. In normal specifications, the designer has to worry about the current object information that is passed to the method and understand how the object's private scope is built.

The semantic function evaluate defines the process of computing the expression result. Here, we present its signature and definition for new instance creation and variable access expressions.

- evaluate _ :: OOExpression → action.

  evaluate ⟦ "new" i:Identifier "(" a:Arguments ")" ⟧=

  ```
      | evaluate arguments a
      then
      | create a new instance of the class bound to i.
  ```

  evaluate ⟦ e:Expression "." i:Identifier ⟧ =

  ```
      evaluate e then give i bound in given object.
  ```

To create a new class instance we just evaluate the arguments passed to the constructor function (operator evaluate arguments) and build a new instance using the operator create a new instance defined by the notation proposed in this paper. To access variables of an object, we just evaluate the expression that indicates the object and use the operator bound in to retrieve the value of the variable. Once again, using the semantic functions defined in this paper, the designer does not need to worry about the object representation.

4.3 Semantic Entities

To use the semantic entities of object-oriented languages we have to define the operators left open.

- type = class | Integer | String.
- allocate for type _ :: type → action.
- allocate for type t = allocate a cell.

In the example we define the sort type and the operator allocate for type. In this case study we do not have to extend the object-oriented semantic entities to express the language constructions.

5 Related works

In the literature we can find many works presenting semantics for object-oriented programming languages. Various kinds of semantic models are used. In [1], Abstract State Machine is used as a technique to model the semantics of a Java subset; the goal is to provide a basis for the standardization of Java and to prove the correctness of an implementation of the Java Virtual Machine. Operational semantics is used in [12,4,15], where the purpose is mainly to prove the type-safety of Java, and in the work reported in [6], which also includes a derivation from the semantics of an interactive programming environment that provides textual and graphical visualization during program execution. In [17] Action Semantics is also used to specify a Java subset. That work helps to validate and to understand the Java official informal specification.

Most of these works consider the same Java subset, which includes primitive types, instance variables and methods, classes with inheritance, interfaces, dynamic binding, overloading and overwriting of methods, instance creation, null pointers, and basic exceptions. In [4] treatment of exceptions is contemplated, as is in [1], which includes static variables, static methods, and concurrency.

Like [17] we also use Action Semantics to specify a subset of Java; we do not treat concurrency, but we include static variables and methods, variable shadowing, method overload, and variable and method accessibility. Moreover, our main goal is to provide a framework to describe the semantics of object-oriented programming languages: not just Java, but also languages like C++ and others. The framework supports all object-oriented concepts mentioned above, and others like multiple inheritance.

6 Conclusions and Future Works

In this work we have used Action Semantics to define semantic entities and operators that facilitate the design of object-oriented programming languages. These semantic entities and operators constitute a library which can be used in the specification of particular object-oriented programming languages.

We have presented a case study based on Java, but the semantic entities and the operators may be used to specify other object-oriented programming languages, like C++. Using the library, the specification of the object-oriented kernel of programming languages like this is greatly simplified. The designer does not have to specify the object-oriented semantic entities and works with a simple notation. The only difficulty left should be the handling of non-object-oriented programming features like pointers.

We independently specified the semantics of a C++ subset [8] and a Java subset [14,7]. It is clear that, with the semantic entities and operators presented here, our work would be easier and the results more readable.

The framework proposed does not support all object-oriented languages yet. Some concepts were not contemplated, as method overloading. The library can also be made more adaptable and extensible to better reflect object-oriented programming language properties. There are object-oriented languages with constructors not described in this paper. Eiffel inheritance with renaming is an example. We can also consider concurrency.

Tools may be designed to generate compilers from descriptions using the proposed framework; [13], for example, reports the generation of a compiler from an Action Semantics description of ADA. This is our next step in this work. This tool will be a further motivation for the development of more cases studies.

Acknowledgement

This work is partially supported by CAPES and CNPq, the Brazilian research agencies.

References

[1] Börger, E. and W. Schulte, *A programmer friendly moduler definition of the semantics of Java*, Technical report, Christien Alberechts Universität Kiel, Intitut für Informatik und Praktische Matematik (1997).

[2] Christensen, S. and M. H. Olsen, *Action semantics of "CCS" and "CSP"*, Internal Report DAIMI IR-82, Computer Science Department, Aarhus University (1988).

[3] de S. Menezes and S. C. B. Soares and J. B. Meneses and H. P. Moura and A. L. C. Cavalcanti, L. C., *A framework for defining object-oriented languages using action semantics (extended version)* (2000), available at `http://www.di.ufpe.br/~rat/papers/oo-framework.ps`.

[4] Drossopoulou, S. and S. Eisenbach, *Towards an Operational Semantics ond Proof of Type Soundness for Java* (1998), department of Computing Imperial College of Scienc, Tecnology and Medicine.

[5] Gosling, J., B. Joy and G. Steele, "The Java Language Specification," Addison-Wesley, 1996.

[6] Isabelle Attali, a. M. R., Denis Caromel, *A formal executable semantics for java*, in: *Proceedings of Formal Underpinnings of Java*, OOPSLA'98, Vancouver, 1990.

[7] Meneses, J. B., *Semântica de Ações de Supernova* (1999), available at `http://www.di.ufpe.br/~rat/asd/supernova-as/supernova.doc`.

[8] Meneses, L. C. S., *SimpleC++ Action Semantics* (1999), available at `http://www.di.ufpe.br/~rat/asd/simplec++-as/simplec++.ps`.

[9] Mosses, P. D., *Unified Algebras and Modules*, Departamental Report DAIMI PB–266, Aarhus University, Computer Science Department, Denmark (1988).

[10] Mosses, P. D., "Action Semantics," Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[11] Mosses, P. D. and D. A. Watt, *Pascal Action Semantics, version 0.6* (1993), available by FTP at `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWatt93DRAFT/pas-0.6.ps.Z`.

[12] Oheimb, D. v. and T. Nipkow, *Machine-cheking the Java Specification: Proving Type-Safety*in: J. Alves-Foss, editor, *Formal Syntaxe and Semantics of Java*, LNCS **1**, Springer, 1998 .
URL `http://www.in.tum.de/~isabelle/bali/doc/Springer98.html`

[13] Palsberg, J., *An Automatically Generated and Provably Correct Compiler for a Subset of Ada*, in: *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, IEEE, 1992, pp. 117–126.

[14] Soares, S. C. B., *Semântica de Ações de Minijava* (1999), available at `http://www.di.ufpe.br/~rat/asd/minijava-as/minijava.doc`.

[15] Syme, D., *Proving JavaS Type Soundness*, Technical Report 427, University of Cambridge Computer Laboratory (1997).

[16] Watt, D. A., *An Action Semantics of Standard ML*, in: *Proc. Third Workshop on Math. Foundations of Programming Language Semantics, New Orleans*, Lecture Notes in Computer Science **298** (1988), pp. 572–598.

[17] Watt, D. A., *Joss Action Semantics, version 1* (1997), available at `www.dcs.gla.ac.uk/~daw/publications/JOOS.ps`.

[18] Watt, D. A. and P. D. Mosses, *Action Semantics in Action* (1987), unpublished.