Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 205 (2008) 123–135
www.elsevier.com/locate/entcs

Aspect-Oriented Action Semantics Descriptions

Luis Menezes

Department of Computing Systems
Universidade of Pernambuco
Recife, Brazil
e-mail: lcsm@dsc.upe.br

Abstract

The modularity of programming language descriptions allows the designer to describe each programming language feature in a separated module that can be studied independently of others. Action Semantics is a formal notation that produce modular descriptions of programming languages. This paper proposes to use concepts of Aspect-oriented programming to improve the modularity of action semantics descriptions. To achieve this goal, an aspect-oriented notation is proposed and applied to describe some programming language concepts.

Keywords: Formal Semantics, Aspect-Oriented Programming, Action Semantics, Programming Languages

1 Introduction

In programming language descriptions, the modularity property allows the designer to isolate a language's feature in a single piece of description. This is important because it facilitates the insertion and removal of these features and their analysis independently of unrelated concepts. Traditional notations used for formal programming language description, such as denotational[10] and operational semantics[3], force the designer to produce programming language descriptions with poor modularity because the definition of one concept may affect the definition of other ones.

Action Semantics is a formal notation designed to describe programming language semantics. Its most important feature is a notation that describes abstractly programming language concepts, hiding their complexity and how programming language concepts affect other ones. This independence leads to the production of more modular and reusable descriptions.

This paper proposes to use aspects to improve the modularity of action semantics descriptions. Aspect-oriented programming is a programming technique designed to isolate the definition of crosscutting concerns (features whose implementation

doi:10.1016/j.entcs.2008.03.069

affects several modules in a system) in a single module, which could be altered or even removed with minimal impact on other modules.

This paper is structured in the following parts:

- Section 2 shows the importance of good modular descriptions in programming language descriptions;
- Section 3 and Section 4 describe the features of action semantics and its object-oriented extensions that lead to more modular descriptions;
- Section 5 contains a case study where the description of the programming language concept affects other specification elements, reducing the modularity of descriptions and Section 6 presents how aspect-oriented programming has been used to solve similar problems in computing systems;
- Section 7 proposes an aspect-oriented action notation to solve the problem presented in Section 5 and applies the notation to some case studies;
- Section 8 presents a formal definition of the notation proposed;
- Section 9 presents the conclusions and suggests future research on the subject.

2 Modularity of Programming Language Descriptions

Modularity is the property of systems that measures the extent to which they have been composed out of separate parts called modules. Each module is responsible for implementing a particular feature of the system, independently of other systems elements. This independence minimizes the effects of inserting, removing or changing modules on the rest of the system and facilitates its manipulation.

In programming language descriptions, modular documents organize the language specification in independent fragments, each one responsible to model some language feature (a command, expression, etc.). This organization is useful to facilitate the analysis of complex modern languages. However, the methodologies more often used to describe language semantics, such as Operational Semantics and Denotational Semantics, produce specifications with poor modularity. The most important modularity problems in these formalisms arise because their descriptions handle directly with the information flow existing in programming languages. This flow is heavily dependent on the language features and may be completely redesigned when the language is changed. For example, when a designer wants to describe the denotational semantics of an expression language and this language is formed only by constants and operations over constants, the semantics is expressed as a value and the evaluation function as:

$$\mathsf{evaluate} :: \mathsf{Expression} \rightarrow \mathsf{Value}.$$

If the expression language contains declared variables, the semantics of expressions should be modeled by a function that receives the current bindings and produces the expression value, as it is shown below:

$$\mathsf{evaluate} :: \mathsf{Expression} \rightarrow \mathsf{Bindings} \rightarrow \mathsf{Value}.$$

If the expression language retrieves and changes the values stored in a memory, the evaluation function becomes:

evaluate :: Expression → Bindings → Store → (Value,Store).

Therefore, each change in the function signatures forces the whole description to be redesigned to handle with the new parameters sent to the function.

3 Action Semantics

Action Semantics [7] is a formalism designed to facilitate the description of programming languages. In order to reach this goal, action semantics has some interesting properties:

- action semantics describes the characteristics of a programming language using a formal notation (action notation) based in terms of the English language, making the specifications more easily understood;
- action semantics allows that specifications can be extended and be reused in new projects of programming languages;
- descriptions in action semantics can be used for automatic generation of implementations using tools such as Actress [9] and Abaco [8].

In action semantics, the meaning of a program is given using predefined semantic entities called *actions* and *yielders*. Actions are dynamic entities that can be executed, producing modifications in the program state. Yielders define expressions whose evaluation results depend on the current state and are used to model computations dependent on that state.

Actions and yielders are used to represent concepts found in traditional programming languages. Their use avoids the designer having to worry about how these concepts are modeled and how they interact with other ones. This property facilitates the design of complex structures and reduces its effects on the rest of the description. For this reason, the equations found in action semantics descriptions become more independent of unrelated features and, therefore, easier to be reused and extended.

The action notation operators are divided into Facets. Each facet describes actions and yielders designed to model some specific programming language feature. The most important Action Notation facets are:

- the basic facet: defines actions that model control flows existing in programming languages (selection, loops, exceptions, etc.);
- the functional facet: defines actions that represent the processing of calculations in programs;
- the declarative facet: defines actions that manipulate scoped declarations in programs;
- the imperative facet: defines actions that manipulate the memory of the program;
- the reflective facet: defines the concept of abstraction that is used to model

procedures and functions.

4 Modular Action Semantics Extensions

Usually, action semantics descriptions are composed of three sections:

- The *Abstract Syntax* describes the structures found in the language using the BNF notation;
- The *Semantic Functions* defines a map between syntax entities and their meaning, which are expressed using the action notation and semantic entities;
- The *Semantic Entities* defines auxiliary types and operations that will be used by semantic functions to describe the meaning of programs.

This structure forces the designer to split a description of each language feature in at least three documents. This makes the identification of the location of the definition of a specific language operator difficult.

To solve this problem, several research works ([2,6], [5] and [1]) propose a new style to describe programming languages. These papers propose that a programming language description should be formed by the union of programming language elements. Each element is a self-contained object with all information necessary to describe it: the syntax, semantic functions and semantic entities. Using the new description style, the designer can define a language feature in a single and independent document section, facilitating its reuse and redesign.

Figure 1 illustrates (using the notation proposed by [5]) the description of a mathematical expression language formed by two different components. Each component definition contains the following properties: the component name; the component class; the component abstract syntax and its semantics. The expression description contains a component Constant that defines a new Expression (Exp). This component syntax is formed by a single number and its meaning is an action that gives this number. The second defined component is the component Sum, which defines another Expression formed by a sequence of: an Expression, the symbol "+" and another Expression. Its semantics consists in computing the sum of the transient value obtained from its sub-expression computations.

5 Descriptions of Syntax-less Language Features

Using the modular description style shown in Section 4, we can isolate the definition of elements with well-defined syntax and semantics. However, programming languages may contain semantic elements with no associated syntax. The existence of these "syntax-less" features affects the semantics of other components, producing poor modular descriptions.

One example of this problem is the description of languages with lazy expressions (a programming language contains lazy expressions if the expressions are evaluated only when the expression value becomes necessary): to give the semantics of lazy expressions the programming language description has to produce a code segment

```
component Constant is Exp where
    syntax =
      ⟦ n:Number ⟧
    semantics =
      give n

component Sum is Exp where
    syntax =
      ⟦ x:Exp "+" y:Exp ⟧
    semantics =
      | | semantics of x
      and then
      | | semantics of y
      then
      | give the sum of them
```

Fig. 1. Component Based Description of an Expression Language

that evaluates the expression value instead of the evaluated value itself. Moreover, before the expression value can be used by the program, this code segment should be executed to provides the expression result. Figure 2 shows the result specification obtained by the redesign of the Expression Language semantics shown in Figure 1. Because the lazy expression semantics modifies the semantics of other components descriptions, it is difficult to isolate its definition.

6 Aspect-Oriented Programming

Aspect-oriented Programming (AOP) [4] is a programming technique designed to modularize the implementation of *Crosscutting Concerns*. A crosscutting concern is a system requirement whose implementation can not be isolated in a single module using traditional programming techniques. Usually, the implementation of a crosscutting concern is fragmented and placed in other system modules originally designed to implement other system requirements.

AOP proposes to model such concerns using the concept of Aspect. An aspect is defined by a sequence of *advices* and *inter-type declarations*. These elements describe points in the system and their modifications. Using aspects, the designer can specify different application points must be changed and how this should be done in order to implement some crosscutting concern. This means that the concern definition becomes separate from the rest of the application, which has its modularity increased.

The implementation of aspects is modeled by the *weaving* operation that produces a new version of the original system with the modifications specified by the desired aspects (and the crosscutting concern implemented).

```
component Number is Exp where
    syntax =
      ⟦ n:Number ⟧
    semantics =
      give abstraction of give n

component Sum is Exp where
    syntax =
      ⟦ x:Exp "+" y:Exp ⟧
    semantics =
      give abstraction of
      | | semantics of x
      and then
      | | semantics of y
      then
      | | enact the given abstraction#1
      and then
      | | enact the given abstraction#2
      then
      | | give the sum of them
```

Fig. 2. Component Based Description of a Lazy Expression Language

The idea of aspects has been successfully used to modularize the implementations of concerns like: Error Handling, Concurrency, Communications, etc.

7 Aspect-Oriented Action Semantics Descriptions

When analyzing the properties and benefits of AOP in system development, its possible to consider whether its good features are helpful to increase the modularity of programming language descriptions. To verify this hypothesis, a set of operators to support an aspect-based style of description with action semantics producing the Aspect-Oriented Action Semantics Descriptions (AOASD) was designed. [1]

In AOASD, aspects are defined from a sequence of advice definitions using the operator 'aspect { a }'. Each advice describes one modification necessary to be performed in the original specification to implement some programming language concept. In the AOASD initial version, the following kinds of advices are defined:

- the advice 'change semantics of x to a', redefines the whole semantics of the components, changing the original semantics x to a;
- the advice 'rewrite a to b', scans the tree of terms in the specification and replaces all terms matching a to b. This replacement affects terms in the specification

[1] To illustrate the proposed notation, this adopts the component-based notation defined in [5] but the same principles could also be adopted in other modular action semantics proposals.

code. Neither terms in b nor terms produced dynamically during the program action execution are modified by this advice.

These advices affect all components in the specification. However, they could be constrained to act on specific components or conditions using the following pointcut operators:

- '**inside** n a': the advice a can be applied only inside the components n, where n can be a specific component name or a class of components;
- 'a **when** c': the advice a can be applied only when the runtime condition c holds. When this point-cut is specified, the modified code should evaluate the condition during the program execution. If it holds the modified action is executed. Otherwise, the original action should be executed. This pointcut is useful to model advices that can be dynamically disabled or depends on some runtime state to be activated.

Finally, the semantics of these aspect operators is provided by the operator '**weaving** d **to** a', that produces a new language description formed by the language description d modified with the changes specified by the aspect a.

Using AOASD, the definition of several programming language concepts may be simplified. Some examples of descriptions using the aspect notation are given in the sections that follow.

7.1 The Lazy Expression Aspect

The semantics of lazy expressions can be modeled using the aspect presented in Figure 3. This aspect contains two advices that redesign the semantics of expressions and the data operation applications.

The first advice changes the semantics of expressions in order to give a function that will give the evaluated value instead of the value itself. When applied to the following component:

```
component Constant is Exp where
    syntax =
      ⟦ n:Number ⟧
    semantics =
      give n
```

The advice will produce the following redesigned component:

```
component Constant is Exp where
    syntax =
      ⟦ n:Number ⟧
    semantics =
      give lazy datum with abstraction of give n
```

The second advice changes the actions that execute data operations in such a way that the lazily evaluated values are calculated before the operator is executed. It

```
aspect {
    inside Exp
    | change semantics of x to
    | | give lazy datum with abstraction of x

    rewrite (give (d:DataOperation)(a:Arguments)) to
    | | give a
    then
    | foreach them do
    | | | give the given eager-evaluated-datum
    | | or
    | | | enact the given lazy-evaluated-datum
    then
    | give d(them)
}
```

Fig. 3. Aspect of Lazy Expressions

captures all actions that give the result of data operations and replace them with new actions that execute non-evaluated values before executing the data operation. For example, this advice would capture the action:

```
give sum(them)
```

and produce the following modified action:

```
| foreach them do
| | | give the given eager-evaluated-datum
| | or
| | | enact the given lazy-evaluated-datum
then
| give sum(them)
```

When these aspects are used, the complexity of the expression semantics remains simple because it avoids replicating the actions that evaluate the lazy values in every specification point that execute data operations.

7.2 The Error of Division By Zero Aspect

Some programming languages handle error situations such as the division by zero raising an exception that can be captured and handled by the program. Action Notation handles this occurrence generating an abnormal execution state that interrupts the execution.

If a language designer wants to specify an alternative handling code, he has to build a new division expression specification with the appropriate error verification

```
aspect {
    rewrite give division(a,b) to
    | escape with ZeroDivisionError
    when (b is 0).
}
```

Fig. 4. Aspect of Division Errors

code for the language described. Another approach to solve this problem is to use an abstract division operator that the designer should define in order to specify the semantics of errors situations. This approach increases the modularity but its frequent use may force the designer to deal with a lot of unnecessary operators, designed to support language features not present in the specified language.

Using AOASD, its possible to model the verification code using the aspect shown in Figure 4. This aspect extends the semantics of all expressions containing the 'division' operator. These occurrences are replaced by the appropriate exception raising action when the divisor becomes zero.

7.3 Aspects for Static Binding and Dynamic Binding

Programming languages procedures may implement two kinds of binding behavior: Static or Dynamic. In a programming language with static bindings, the procedure execution uses the scoped information active when the procedures are *defined*. In a programming language with dynamic bindings, the scoped information used by procedures is the scope active when the procedure is *executed*.

The action notation defines three operators designed to handle procedures and their scopes:

- the operator 'abstraction of *ac*' defines abstractions. An abstraction is a value that encapsulates the computation defined by the action *ac*;
- the action 'enact *ab*' executes the computation encapsulated by the abstraction defined by *ab*;
- the operator 'closure *ab*' defines an abstraction that encapsulates the current scope information in the abstraction defined by *ab*.

The semantic difference between statically and dynamically bound languages is modeled by the location where the 'closure' operation is executed. Using the existing styles of action semantics descriptions, the designer has to build two different versions for components that describe the execution and the definition of procedures. If the designer adopts AOASD, he just specifies these components with no indication about when the closure is performed and applies either the aspect 'Static Binding' (Figure 5) or the aspect 'Dynamic Binding' (Figure 6) to the language definition. These aspects are responsible for putting the 'closure' operator in the appropriate location.

aspect {
rewrite abstraction of a to closure abstraction of a.
}

Fig. 5. Aspect of Statically Bound Languages

aspect {
rewrite enact a to enact closure a.
}

Fig. 6. Aspect of Dynamically Bound Languages

- aspect _ :: advice* $\rightarrow$ aspect.
- inside _ _ :: identifier, advice $\rightarrow$ advice.
- when _ _ :: yielder, advice $\rightarrow$ advice.
- change semantics from x to y :: term, term $\rightarrow$ advice.
- rewrite _ to _ :: term, term $\rightarrow$ advice.
- _ weaving _ :: language-description, aspect $\rightarrow$ language- description.

Fig. 7. Signatures of the Operators Defined in the AOASD

8 Formal Description of AOASD

This section proposes a rewriting semantics for AOASD. The signatures of the defined operators are shown in Figure 7. The following rules describe the semantics of these operators.

- (c_1 c_2) weaving a = (c_1 weaving a) (c_2 weaving a).
- c weaving (aspect a_1 a_2) =
  (c weaving aspect a_1) weaving aspect a_2 .

The weaving operation, when applied to composed components or aspects, can be modeled as the composition of simpler weavings. The following rule describes the functionality of the advice constraint 'inside':

- c=(component n of t is (syntax syn) (semantic sem)) $\Rightarrow$

c weaving (aspect inside p a) =
    if either(p is n, p is t) then
        c weaving (aspect a)
    else
        c

When an advice constrained to a specific component is weaved to other component, the weaving operation checks if this component is compatible with the advice itself. If the test succeeds, the weaving operation is performed. Otherwise, the weaving operations make no modifications in the component. The next rules describe the semantics of 'when' operator:

- (change semantics from a to b) when c =
    change semantics from a to
    | check c and then b
    or
    | check not (c) and then a
- (rewrite a to b) when c =
    rewrite from a to
    | check c and then b
    or
    | check not (c) and then a

The operator 'when' is used to specify runtime conditions that indicate when the transformed action should be executed instead of the original action. Depending on this condition, the original action or the replacement will be performed. The next rules are intended to describe the rewriting semantics of weaving:

- (*flag*,*binds*)=match(a,se) ⇒
    c = (component n is t where syntax sy semantics se) ⇒
        c weaving (aspect change semantics from a to b)=
            if (*flag*) then
            | (component n is t where syntax sy semantics b[*binds*])
            else
            | c.

The weaving of unconstrained 'change semantic' advices verifies if the component semantics matches the left-side term. If this matching fails, the component is kept unchanged. Otherwise, the component semantics is replaced by the advice right-side term applied to the variable bindings produced by the matching operations.

- c=(component n is t where syntax sy semantics se)⇒

```
c weaving (aspect rewriting a to b )=
    component n is t where
        syntax =
          sy
        semantics =
          apply [a ⇒ (nonrecursive b)] to se
```

The weaving of a rewriting advice just modifies the component's semantics using the standard term rewriting semantics.

The operator '$\mathsf{match}(x,y)$' uses a pattern-matching algorithm to compare the terms x and y, it returns a pair representing the matching result and the bindings of the variables in x calculated by the matching operation. The operator '$t[b]$' produces a term t, modified according the variable instantiations specified by the bindings b. The operator '$\mathsf{apply}\ r\ t$' applies the rewriting rule r in the term t. The operator $\mathsf{nonrecursive}\ b$ is used to specify that the rewriting engine should not process the term inside b.

9 Conclusion and Future Works

This paper proposes an aspect-oriented notation for action semantics descriptions. AOASD is useful to increase the modularity and reduce the complexity of action semantics descriptions as shown in case studies. These results stimulate further investigation on the study of the descriptions of complex languages using aspects and provide supporting tools for AOASD. In addition, extensions to incorporate new aspect-oriented concepts such as inter-type declarations should be inserted in AOASD to improve the power of this technique.

Furthermore, we think that these ideas could be useful to increase the modularity of other formal methods.

References

[1] Carvilhe, C. and M. A. Musicante, *Object-oriented action semantics specifications*, Journal of Universal Computer Science **9** (2003), pp. 910–934.

[2] Doh, K.-G. and P. D. Mosses, *Composing programming languages by combining action-semantics modules*, Sci. Comput. Program. **47** (2003), pp. 3–36.

[3] Kahn, G., *Natural semantics*, in: F. J. Brandenburg, G. Vidal-Naquet and M. Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science* (1987), pp. 22–39.

[4] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, Proceedings European Conference on Object-Oriented Programming **1241** (1997), pp. 220–242.

[5] Menezes, L. C. and H. Moura, *Component-based action semantics: A new approach for programming language specifications*, in: *SBLP 2001 - V Brazilian Symposium on Programming Languages*, 2001, pp. 152–163.

[6] Mosses, P., *A constructive approach to language definition*, Journal of Universal Computer Science **11** (2005), pp. 1117–1134.

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

[8] Moura, H. and L. Menezes, *The abaco system: An algebraic based action compiler*, in: *Proceedings of the Second International Workshop on Action Semantics*, number NS-99-3 in BRICS Notes Series, 1999, pp. 143–154.

[9] Moura, H. and D. A. Watt, *Action transformations in the* ACTRESS *compiler generator*, CC94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh **786** (1994), pp. 16–30.

[10] Schmidt, D. A., "Denotational Semantics," Allyn & Bacon, 1986.

A Component Based Action Semantics

This section presents the signature of the component operator used in this paper.

language-definition = component*.

A component-based language definition is formed by several programming language components;

component _ is _ where syntax = _ semantics =_ ::
identifier, identifier, syntax-tree, action.

A component definition contains the following elements: the component name, the component type name, a syntax tree that describe the component elements and an action that provides the component meaning.

⟦ _ ⟧ :: syntactical-element* → syntax-tree.

string ≤ syntactical-element.

_ : _ :: variable-name, identifier → syntactical-element.

A syntax-tree used by component notation is formed by a sequence of syntactical elements delimited by braces. A syntactical element can be either: a string or a reference of a syntactical class of elements.

B Auxiliary Notation

This section describes the semantics of auxiliary actions used in this paper. These actions do not belong to standard action notation definition but they are useful to simplify the resulting description.

foreach _ do _ :: data, action → action.

foreach (x,y) do a =
  | foreach x do a
  and then
  | foreach y do a.
foreach x:datum do a = give x then a.

The action foreach t do a receives a tuple of values (t) and executes the action (a) for each value belonging to this tuple.