Journal of Universal Computer Science, vol. 13, no. 6 (2007), 891-919
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

Using Visitor Patterns in Object-Oriented Action Semantics

André Murbach Maidl
(Federal University of Paraná, Brazil
murbach@inf.ufpr.br)

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

Martin A. Musicante
(Federal University of Rio Grande do Norte, Brazil
mam@dimap.ufrn.br)

Abstract: Object-Oriented Action Semantics is a semantic framework for the definition of programming languages. The framework incorporates some object-oriented concepts to the Action Semantics formalism. Its main goal is to obtain more readable and reusable semantic specifications. ObjectOriented Action Semantics provides support for the definition of syntax-independent specifications, due to the way its classes are written. In a previous work, a library of classes (called LFL) was developed to improve specification reuse and to provide a way to describe semantic concepts, independent from the syntax of the programming language. This paper aims to address some problematic aspects of LFL, and presents a case study, where a specification is built by using the Visitor Pattern technique. The use of this pattern allows a clear separation between the syntax of a programming language and its different semantic aspects.

Key Words: formal semantics, action semantics, object-oriented specifications.
Category: F.3.2, D.1.5

1 Introduction

Action Semantics [Mosses 1992, Watt 1991] is a formal framework for defining the semantics of programming languages. A main goal of Action Semantics is to provide a notation that is intuitive for programmers. This notation has been used to describe the semantics of real programming languages and presents good reusability and extensibility properties [Mosses and Musicante 1994]. However, the standard Action Semantics lacks of some syntactic support for the definition of libraries and reusable components [Gayo 2002].

Several approaches that introduce modularity to Action Semantics have been proposed. Modular Action Semantics is presented in [Doh and Mosses 2001]. Their goal is to extend the Action Semantics notation to define modules. The interpretation of modules introduces some new, potential, problems to the semantics of the whole specification [Doh and Mosses 2001]. These problems are related to the way the modules and their hierarchy are interpreted.

Based on the modular approach, an object-oriented view of Action Semantics is proposed in [Carvilhe and Musicante 2003]. This approach is called Object-Oriented Action Semantics (OOAS). Using OOAS it is possible to organize Action Semantics descriptions into classes. OOAS does not have the problems of Modular Action Semantics since the class hierarchy is taken into account for the interpretation of a semantics description.

The **LFL** (Language Features Library) is a library of OOAS classes, proposed in [Araújo and Musicante 2004]. Its goal is to aid in the creation of new OOAS descriptions by composing and reusing classes. The main goal of LFL is to define generic classes to describe general semantic concepts. These classes can be instantiated to be used in conjunction to the syntax of a specific programming language. However, LFL introduced some syntactic problems to Object-Oriented Action Semantics descriptions. In particular, the way in which generic classes were defined is cumbersome, resulting in less readable specifications.

In this work, we propose LFLv2 (read LFL version 2), a new version of LFL to solve the problematic issues in the original library. Moreover, we incorporate in LFLv2 some of the ideas presented in [Mosses 2005, Iversen and Mosses 2005], in order to separate the syntax of the programming language from the specification of its semantics. In addition to it, we give examples of Object-Oriented Action Semantics descriptions using the Language Features Library version 2 and also using the Visitor Pattern technique [Gamma et al. 1998, Appel and Palsberg 2003] to improve the modularity aspects observed in the object-oriented approach.

This work is organized as follows: the next section briefly introduces Action Semantics and OOAS. Section 3 sumarizes the LFL. In section 4 we introduce the use of the Visitor Patterns technique, as well as LFLv2. A simple imperative language is specified as a case study in section 5. Section 6 gives the conclusions of this work.

2 Action Semantics and OOAS

Action Semantics [Mosses 1992, Watt 1991] is a formal framework developed to improve the readability and the way of describing programming languages semantics. The framework is based on Denotational Semantics and Operational Semantics. In Action Semantics, semantic functions specify the meaning of the phrases of a language using *actions*. These actions represent the denotation of the language phrases. The Action Notation is defined operationally and contains the actions and action combinators needed in Action Semantics descriptions.

The three main mathematical entities that compose Action Semantics descriptions are: *actions*, *data* and *yielders*. Actions can be performed to represent the concepts of many programing languages, such as: control, data flow, scopes of bindings, effects on storage and interactive process. When an action is performed it can produce one of the following outcomes: normal termination (*complete*), exceptional termination (*escape*), unsuccessful termination (*fail*) or non-termination (*diverge*). Action Notation provides some basic actions which are written using English words in order to improve readability in semantics specifications. These actions can be combined to obtain more complex actions.

A data notation is used in Action Semantics to provide the needed sorts of data and operations related to them. These sorts of data can be: truth-values, numbers, characters, strings, tuples, maps, tokes, messages, etc.

Yielders are entities that can produce data during an action performance. Such produced data are called *yielded data*. The produced data depend on the current information processed, for instance: the *transient data* passed, the *bindings* received and the current *storage* state. Actions have *facets*, according to their data flow:

– **basic** - deals with pure control flow;

– **functional** - deals with actions that process transient data;

– **declarative** - deals with actions that produce or receive bindings;

– **imperative** - deals with actions that manipulate the storage;

– **reflective** - deals with abstractions;

– **communicative** - deals with actions that operate under distributed systems.

Object Oriented Action Semantics (OOAS) [Carvilhe and Musicante 2003] is a method to organize Action Semantics specifications. Its main goal is to provide extensibility and reusability in Action Semantics by splitting semantic descriptions into classes and treating semantic functions as methods.

In order to overcome the lack of modularity in Action Semantics, as reported by [Gayo 2002], Object-Oriented Action Semantics introduces an extension of the standard Action Notation. Class constructors and several other object-based operators are available, to provide an object-oriented way of composing specifications. An Object-Oriented Action Semantics description is a ***hierarchy*** of classes. Each class encapsulates some particular Action Semantics features. Those features can be easily reused or specialized by other objects, using well-known object-orientation concepts. As an example, we will take a look at a simple command language, whose BNF is defined as follows:

```
Command ::= Identifier ":=" Expression | Command ";" Command |
            "if" Expression "then" Command 'else" Command "end-if" |
            "while" Expression "do" Command
```

This equation defines syntactic trees for commands, containing assignments, sequences, conditionals and iterations. A simple strategy for defining an Object-Oriented Action Semantics description of this language is to define Command as an abstract class, and to define sub-classes of commands for each class of phrase. A base-class can be written as:

```
Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Cmd → Action
End Class
```

Notice that Command is the abstract class. Initially it introduces the syntactic sort Cmd, detailed in the syntax section. Semantics part defines the semantics of a Command, using standard Action Notation. In this case, it establishes that a semantic function execute maps a Command to an Action. A plain semantic function like execute is seen as a method.

The Command definition provides a foundation to specify the other classes. We can now define every particular class as a sub-class of Command. Lets take a look at the Selection class as follows:

```
Class Selection
    extending Command
    using $E$:Expression, $C_1$:Command, $C_2$:Command
    syntax:
        Cmd ::= “if” $E$ “then” $C_1$ “else” $C_2$ “end-if”
    semantics:
        execute ⟦ “if” $E$ “then” $C_1$ “else” $C_2$ “end-if” ⟧ =
            evaluate $E$ then execute $C_1$ else execute $C_2$
End Class
```

Selection is the specialized class. The extending directive states a particular Command behavior (in this case, a conditional command). The using directive makes other objects available.

A complete Object Oriented Action Semantics specification can be found in [Carvilhe and Musicante 2003].

3 Language Features Library (LFL)

In programming languages specifications it is common to find pieces of the specification that have similar concepts. When it happens is interesting to reuse these pieces instead of having duplicated code. The classes concept introduced by Object-Oriented Action Semantics [Carvilhe and Musicante 2003] helps with code reuse due to the common structure provided by organizing the specification into classes.

In [Araújo and Musicante 2004], was proposed a library of classes which was called **LFL** (Language Features Library). The function of LFL is to congregate and organize classes, that have common specifications, into a structure and work as a repository of classes for Object-Oriented Action Semantics descriptions.

A tree structure was adopted to represent the classes' organization in LFL. LFL was branched off into three main classes: *Syntax*, *Semantics* and *Entity*.

The node *Semantics* was the only one implemented in the initial version of LFL and it also forked into another three nodes: *Declaration*, *Command* and *Expression*. Pieces of code that manipulate bindings are treated in *Declaration*; *Command* represents semantic definitions that are concerned about data flow and the storage; while those classes for the manipulation of values belong to *Expression*.

Each one of the above-defined nodes was split into two: *Paradigm* and *Shared*. The first one is responsible to represent classes that contain features from a specific programming language paradigm, such as: *Object-Oriented*, *Functional*, *Logical* and *Imperative*. The node *Shared* contains classes which represent features that are common to more than one programming paradigm.

The following example illustrates the definition of a LFL class. This class defines the semantics of an if-then-else command:

```
Class Selection
            ≪ Command implementing < execute _ : Command → Action >
            Expression implementing < evaluate _ : Expression → Action > ≫
    locating LFL.Semantics.Command.Shared
    using $E$:Expression, $C_1$:Command, $C_2$:Command
    semantics:
            execute-if-then-else($E$, $C_1$, $C_2$) =
                    evaluate ⟦ $E$ ⟧ then
                    | check(the given TruthValue is true) and then execute ⟦ $C_1$ ⟧
                    or
                    | check(the given TruthValue is false) and then execute ⟦ $C_2$ ⟧
End Class
```

The Selection class provides the semantics of a selection command in a syntax-independent style. The classes Command and Expression are passed as (higher order) parameters. Notice that the generic arguments must comply with the restrictions defined by the implementing directive: They must provide some methods with a specific type as sub-parameters.

The directive locating identifies the place where the class is located in the LFL structure.

The programming language specifications which use the LFL are similar to the plain Object-Oriented Action Semantics descriptions. Let us now exemplify the use of the generic Selection class to define the semantics of a selection command:

```
Class MyCommand
    syntax:
        Com
    semantics:
        myexecute _ : Com → Action
End Class

Class MySelection
    extending MyCommand
    using $E$:MyExpression, $C_1$:MyCommand, $C_2$:MyCommand,
        objSel:LFL.Semantics.Command.Shared.Selection ≪
            MyCommand<myexecute>, MyExpression<myevaluate> ≫
    syntax:
        Com ::= “if” $E$ “then” $C_1$ “else” $C_2$ “end-if”
    semantics:
        myexecute ⟦ “if” $E$ “then” $C_1$ “else” $C_2$ “end-if” ⟧ =
            objSel.execute-if-then-else($E$, $C_1$, $C_2$)
End Class
```

The classes MyCommand and MySelection, as defined above, specify the behaviour of a selection command using the LFL. Notice that the LFL generic class Selection is instantiated. The argument for this LFL class are the MyCommad and MyExpression classes. Notice that the sub-parameters are also provided.

LFL brings an interesting concept to Object-Oriented Action Semantics: the definition of semantic descriptions which are independent from the syntax of the programming language. This definition uses the inclusion of classes provided by a library of classes. A main issue with the way in which LFL is defined is that its parameter-passing mechanism is obscure and let us back to the readability problems found in other semantic descriptions of programming languages.

4 Visitor Patterns

In the previous sections we have presented Object-Oriented Action Semantics, an approach for language definition using Action Semantics, and object-oriented concepts. We also have presented LFL, a library of classes for Object-Oriented Action Semantics descriptions. The former is based on the modularity in Action Semantics by splitting descriptions into classes, the latter is a collection of Object-Oriented Action Semantics classes to be used and instancied in different programming languages projects.

As it is proposed in [Mosses 2005, Iversen and Mosses 2005], we are looking for approaches that allow us to describe programming languages semantics syntax independently. LFL clearly does it, however it has a main problematic issue, related to the way instantiation of classes is done.

In this work, we propose some changes in LFL for its improvement, namely, using LFL with no parameters passing. Visitor Patterns [Gamma et al. 1998] are added to OOAS descriptions that use LFL in order to try to increase the modularity aspects observed in the object-oriented framework. Visitor Patterns are used to define operations that can be performed in the elements of an object structure. They allow the definition of a new operation without changing the classes of the elements in which these operate.

4.1 Object-Oriented Action Semantics and Visitor Patterns

Visitor Pattern is a common object-oriented technique used in compiler construction. Visitor Pattern helps the compiler writer to provide several semantics to the same syntactic tree.

We can use the Visitor Pattern technique to give an interpretation to each syntax tree in OOAS. This interpretation is an (OOAS) object, which has a *visit* method for each syntax tree defined. Each Object-Oriented Action Semantics class should implement an *accept* method to serve as a hook for all interpretations. When an *accept* method is called by a visitor, the correct *visit* method is invoked. Such control can go back and forth between visitors and classes [Appel and Palsberg 2003].

Informally, when the visitor calls the *accept* method it is in fact asking: "who are you?". The question is answered by the *accept* method as a result of the calling of the correspondent *visit* method of the visitor. The following examples illustrate how to use Object-Oriented Action Semantics with the Visitor Pattern. Notice that each *accept* method takes a visitor as a parameter and that each *visit* method takes a syntax tree object as a parameter.

```
Class Command
    syntax:
        Cmd
    semantics:
        accept _ : Visitor → Action
End Class

Class Selection
    extending Command
    using $E$:Expression, $C_1$:Command, $C_2$:Command
    syntax:
        Cmd ::= "if" $E$ "then" $C_1$ "else" $C_2$ "end-if"
    semantics:
        accept $V$:Visitor = visit $V$
End Class
```

```
Class Visitor
    syntax:
        Vis ::= Command
    semantics:
        visit _ : Command → Action
End Class

Class Interpreter
    extending Visitor
    semantics:
        visit ⟦ “if” $E$ “then” $C_1$ “else” $C_2$ “end-if” ⟧ =
            evaluate $E$ then accept $C_1$ else accept $C_2$
End Class
```

Notice that a *visitor* is in fact a Command syntax tree object encapsulated by the Visitor class. When the accept method is performed, the correct visit method is called to interpret the syntax tree argument that actually is a *visitor*. In section 2 the interpreter was implemented in the execute methods while now it is in the Interpreter class. The evaluate method is still used in the traditional way of Object-Oriented Action Semantics.

With the Visitor Patterns we can add new interpretations without editing existing classes since that each class has its own accept method. In section 5 we will see that accept and visit methods can be used with different names when we need more than just one visitor, as was used in this section.

4.2 LFLv2 and Visitor Patterns

LFL is a good approach to be used as a programming language semantics description tool for specifying programming languages semantics syntax independently since it has support for generic classes definitions [Araújo and Musicante 2004]. Nonetheless, the obscure LFL syntax took us back to the readable problems found in semantic frameworks. An alternative way to inhibit parameters passing in LFL is the use of abstractions of actions to specify the semantics of some common concepts of programming languages.

Furthermore, we also propose the LFL usage just for semantic concepts. It implies in the exclusion of the LFL nodes: *Syntax*, *Semantics* and *Entity*. Thereby, we can bind the nodes *Declaration*, *Command* and *Expression* directly to the main LFL class since we will use LFL just to represent semantic concepts. The nodes *Shared* and *Paradigm* are maintained, as well as their respective subdivisions. The former is concerned with constructs that commonly appear in programming languages and the latter represents some specialized constructs that appear in specific programming paradimgs.

As in the first version of the library, the set of classes is very reduced and can be extended to support new classes. According to our proposal, the Selection class example shown in section 3 would be written in the following way:

```
Class Selection
    locating LFL.Command.Shared.Selection
    using: $Y_1$:Yielder, $Y_2$:Yielder, $Y_3$:Yielder
    semantics:
        if-then-else($Y_1$,$Y_2$,$Y_3$) = enact $Y_1$ then enact $Y_2$ else enact $Y_3$
End Class
```

Now, LFL classes are more compact since they just define their own location in the LFL hierarchy, the methods that describe the semantics of some concepts of programming languages and the objects used in the specified methods.

Instead of using methods that have to be passed as parameters, each method implemented by the new library of classes receives a number of abstractions that are performed according to the language behavior that is being represented. The enact action receives *yielders* that result in abstractions (and encapsulate actions). These actions are performed independently from the methods described in the programming language specification. Notice that the use of the Visitor Pattern technique is something that is not tied to the new library. Althoug it has been used since it might improve modularity in OOAS descriptions that use LFL. Let us now see how to use the new LFL with the Visitor Pattern:

```
Class MyCommand
    syntax:
        Com
    semantics:
        myexecute _ : Visitor → Action
End Class

Class MySelection
    extending MyCommand
    using $E$:MyExpresssion, $C_1$:MyCommand, $C_2$:MyCommand
    syntax:
        Com ::= “if” $E$ “then” $C_1$ “else” $C_2$ “end-if”
    semantics:
        myexecute $V$:Visitor = visit $V$
End Class

Class Visitor
    using $O$:LFL
    syntax:
        Vis ::= MyCommand
    semantics:
        visit _ : MyCommand → Action
End Class
```

```
Class Interpreter
    extending Visitor
    semantics:
        visit ⟦ "if" $E$ "then" $C_1$ "else" $C_2$ "end-if" ⟧ =
            $O$.Command.Shared.Selection.execute-if-then-else(
                        closuse abstraction of (myevaluate $E$),
                        closuse abstraction of (myexecute $C_1$),
                        closuse abstraction of (myexecute $C_2$) )
End Class
```

The classes are again defined as in Object-Oriented Action Semantics using Visitor Patterns. The language syntax and the semantic function myexecute are also defined in function of the visitors approach. In this way, the semantics descriptions by the library usage are given in the visitor implementation.

The object O, defined in the Visitor class, represents all LFL classes and they can be used due to the hierarchy of classes created by the library. The visit method describes the semantics of the selection command used in the language passing the abstractions to the execute-if-then-else method specified in LFL.

The specification above defines a language with static bindings. If the language being described has dynamic bindings, the directives *closure* should be taken from the specification.

The notation encapsulate X will be used in the rest of this paper as an abbreviation of closure abstraction of (X).

5 A case study

In this section we present the specification of a toy language called μ-Pascal. This language is fairly similar to the Pascal language. μ-Pascal is an imperative programming language containing basic commands and expressions. The μ-Pascal language will be specified using Object-Oriented Action Semantics, the new LFL proposed in section 4.2 and using the Visitor Pattern technique.

5.1 Abstract syntax

Now we present the abstract syntax for the μ-Pascal language:

```
(1) Program ::= "begin" Declaration ";" Command "end"
(2) Declaration ::= "var" Identifier ":" Type ⟨= Expression⟩? |
                    "const" Identifier ":" Type "=" Expression |
                    Declaration ";" Declaration
(3) Type ::= "boolean" | "integer"
(4) Command ::= Identifier ":=" Expression | Command ";" Command |
                "if" Expression "then" Command ⟨ "else" Command⟩? "end-if" |
                "while" Expression "do" Command
(5) Expression ::= "true" | "false" | Numeral | Identifier |
                   Expression ⟨ "+" | "-" | "*" | "<" | "=" ⟩ Expression
```

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

In the above abstract syntax we have defined that a Program is a sequence of Declaration and Command. Declarations can be integer or boolean constants and variables. Assignments, sequences, selections and iterations are defined as Commands. Expressions might be arithmetical or logical.

5.2 μ-Pascal semantics using LFLv2 and Visitor Patterns

In this section we will demonstrate the use of the Object-Oriented Action Semantics approach with the advantages of the new LFL and the Visitor Pattern technique.

First of all, we will define the basic classes that implement some particular concepts of the example language, like identifiers, numerals and types. After that we will exemplify how declarations, commands and expressions are treated. Then we will show how the *visitors* can join all together, giving the semantics of the language.

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

The class Identifier has just the syntactic part since it is used only for representing the name of variables and constants used in the language.

```
Class Numeral
    syntax:
        N ::= digit+
    semantics:
        valuation _ : N → integer
End Class
```

In the class Numeral we define what kind of numbers are used by the example language. In this case the numbers are integers and the method valuation is defined to map a numeral received as a parameter to its respective integer. Notice that the valuation parameter is a syntactic entity.

```
Class Type
    syntax:
        T ::= “boolean” | “integer”
    semantics:
        sort-of _ : T → Sort
        sort-of ⟦ “boolean” ⟧ = truth-value
        sort-of ⟦ “integer” ⟧ = integer
End Class
```

The Type class is used to define that the language data types can be boolean or integer. The method sort-of maps the language data types to their correct Object-Oriented Action Semantics *sorts*.

Hence, we will see the methods parameters as *visitors* that carries syntactic entities. In fact, the syntactic entities are encapsulated by the *visitor* objects and they will be interpreted by the *visitor* which implements the semantics of the encapsulated syntax.

```
Class Declaration
    syntax:
        Dec
    semantics:
        elaborate _ : Visitor → Action
End Class
```

The class Declaration works as an abstract class. It introduces the sort Dec which will be redefined in the sub-classes to represent each Declaration. Both methods accept and visit, from section 4, are represented by elaborate and visitDec. Notice that Declaration is just an abstract class, reason why visitDec appears just in its sub-classes.

```
Class Variable
    extending Declaration
    using $I$:Identifier, $E$:Expression, $T$:Type
    syntax:
        Dec ::= “var” $I$ “:” $T$ [ “=” $E$ ]
    semantics:
        elaborate $V$:Visitor = visitDec $V$
End Class
```

Constructing the declarations classes hierarchy, now we have defined the Variable class. The Dec token is redefined giving the variables declarations syntax used in the language. The elaborate method is overloaded to express the semantics of a variable declaration using the visitDec method.

The method elaborate takes a *visitor* as an argument and through the method visitDec gives the declaration performance in the Visitor class. The declaration is possible to be performed in the Visitor class since V represents the encapsulated syntax. In this manner, the syntax carried by the *visitor* object can be checked by the correct visit method.

```
Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Visitor → Action
End Class
```

We now have the Command class. In this class we introduce the syntactic sort Cmd which will be redefined in Command sub-classes. In commands we will define the accept method as execute and the visit method as visitCmd. Notice that, like Declaration, Command also is an abstract class.

```
Class While
    extending Command
    using $C$:Command, $E$:Expression
    syntax:
        Cmd ::= “while” $E$ “do” $C$
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class
```

In the While class we express how a while-loop works in the language. To achieve the before mentioned result, we have created the super-class Command and the sub-class While. Using Visitor Patterns, in the abstract class we specify just the abstract method which defines that a *visitor* results in an *action*. A command is correctly executed by calling the execute method with the command syntax. The correct semantics will be given by the visitCmd method since it takes the *visitor* object and interprets the syntax, that is in the object, in the Visitor class.

```
Class Expression
    syntax:
        Exp
    semantics:
        evaluate _ : Visitor $\rightarrow$ Action
End Class
```

Again we have an abstract class, like Declaration and Command. The Expression class is the super-class for the expressions definitions. The accept and visit methods will be represented by evaluate and visitExp, respectively. The sub-classes of Expression can be defined similarly to Declaration and Command sub-classes definitions. Now we will see how the Visitor class works.

```
Class Visitor
    using: $O$:LFL
    syntax:
        Vis ::= Declaration | Command | Expression
    semantics:
        visitDec _ : Declaration $\rightarrow$ Action
        visitCmd _ : Command $\rightarrow$ Action
        visitExp _ : Expression $\rightarrow$ Action
End Class
```

In the Visitor class we specify all the signatures of the visit methods which represent the maps of a class, containing a syntactic tree that will be visited, to an action. We also specify that a Visitor may carry a Declaration, a Command or an Expression syntax tree object. The Object-Oriented Action Semantics descriptions are given in the Interpreter class using the methods provided by **LFL**. The LFL methods are accessed through the object O which is a LFL instance.

```
Class Interpreter
    extending Visitor
    semantics:
        visitDec ⟦ "var" $I$ ":" $T$ ⟧ =
            O.Declaration.Paradigm.Imper.VarDec.elaborate-variable(
                    $I$, sort-of $T$)
        visitDec ⟦ "var" $I$ ":" $T$ "=" $E$ ⟧ =
            $O$.Declaration.Paradigm.Imper.Variable.elaborate-variable(
                    $I$, sort-of $T$, encapsulate evaluate $E$)
        ...
        visitCmd ⟦ $C_1$ ";" $C_2$ ⟧ =
            $O$.Command.Shared.Sequence.execute-sequence(
                    encapsulate execute $C_1$, encapsulate execute $C_2$)
        ...
        visitCmd ⟦ "while" $E$ "do" $C$ ⟧ =
            $O$.Command.Shared.While.execute-while(
                    encapsulate evaluate $E$, encapsulate execute $C$)
        ...
        visitExp ⟦ $N$ ⟧ =
            give valuation $N$
        visitExp ⟦ $I$ ⟧ =
            $O$.Expression.Shared.Identifier.evaluate-identifier($I$)
        visitExp ⟦ $E_1$ "+" $E_2$ ⟧ =
            $O$.Expression.Shared.Sum.evaluate-sum(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
        ...
End Class
```

Notice that the *visitors* visit a syntax tree, specified in the Object-Oriented Action Semantics classes, and give the correct semantics to the syntax tree visited. It is possible by calling the accept method, this method takes a *visitor* represented by a syntax tree object and then the correct visit method will be performed to give the semantics of the syntax tree carried by the *visitor*.

The semantics expressed by the visit methods might be merely an *action*, like in the numeral valuation, just a LFL method that results in an *action*, like in the identifiers evaluation, or we can use accept calls to visit the needed syntactic entities to be used with LFL methods; in the example the *actions* are encapsulated for generating *abstractions* and these *abstractions* are passed to some LFL method that implements the desired semantics. We can also pass *tokens* and *sorts* to the new LFL [Maidl 2007].

```
Class Micro-Pascal
    using $D$:Declaration, $C$:Command
    syntax:
        Prog ::= "begin" $D$ ";" $C$ "end"
    semantics:
        run _ : Prog → Action
        run ⟦ "begin" $D$ ";" $C$ "end"⟧ = elaborate $D$ hence execute $C$
End Class
```

Since Object-Oriented Action Semantics allow us to organize the specification into classes, we can define a main class. The main class in this example is the Micro-Pascal class. In this class we specify the syntax of a μ-Pascal program and that it is mapped to an action. The declaration semantics and command semantics are given by calling their accept methods and passing their respective *visitors*.

The complete case study can be cheked in the appendix B. A version of it using just plain OOAS and Visitor Patterns is also provided in the appendix A.

6 Conclusions

Object-Oriented Action Semantics was designed to improve modularity in Action Semantics descriptions. LFL has improved the reusability in the object-oriented approach and also has provided a way to describe the semantics of programming languages in a syntax-independent style. The problems found in the first version of LFL were solved in the new version of it, through the use of abstractions of actions, instead of passing classes and methods as arguments to LFL classes. The library of classes was also restructured to implement only those pieces that are concerned with representing the semantics of programming languages.

LFL became more concise and simplified due to the use of the reflective facet. The use of the Visitor Pattern improves modularization in OOAS. Notice that this pattern can be used either with just plain OOAS or with OOAS plus LFL. Since the definition of the Visitor Pattern is inherently related to the object-oriented paradigm, the use of this pattern in conjunction with an object-oriented specification language, like OOAS, leads to clear and modular specifications.

In [Maidl 2007], the author proposes an implementation of Object-Oriented Action Semantics in Maude and implements LFLv2 as a case study of it.

As future work, we would investigate the use of the Visitor Pattern technique for compiler generation from OOAS descriptions and we also would trace a careful comparsion between LFLv2 and the constructive approach [Mosses 2005, Iversen and Mosses 2005].

The main contributions of this work can be summarized as:

– Modularity is improved. The use of Object-Oriented Action Semantics and Visitor Patterns provide a new view of modularity in semantic descriptions of programming languages.

– LFL is patched. We propose a cleaner and usable library of classes by its own restructuration and for the fact of changing arguments by abstractions.

– LFLv2 can help to describe programming languages syntax-independently. The new hierarchy of classes made this possible.

Acknowledgments

We would like to thank the anonymous referees for their constructive comments and also Flávia Erika Shibata for her careful English review.

References

[Appel and Palsberg 2003] Appel, A. W. and Palsberg, J. (2003). *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA.

[Araújo and Musicante 2004] Araújo, M. and Musicante, M. A. (2004). Lfl: A library of generic classes for object-oriented action semantics. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile*, pages 39–47. IEEE Computer Society.

[Carvilhe and Musicante 2003] Carvilhe, C. and Musicante, M. A. (2003). Object-oriented action semantics specifications. *Journal of Universal Computer Science*, 9(8):910–934.

[Doh and Mosses 2001] Doh, K. and Mosses, P. (2001). Composing programming languages by combining action semantics modules. In *First Workshop on Language Descriptions, Tools and Applications*.

[Gamma et al. 1998] Gamma, E., Vlissides, J., Johnson, R., and Helm, R. (1998). *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Gayo 2002] Gayo, J. E. L. (2002). Reusable semantic specifications of programming languages. In *SBLP 2002 - VI Brazilian Symposium on Programming Languages*.

[Iversen and Mosses 2005] Iversen, J. and Mosses, P. D. (2005). Constructive action semantics for core ML. *IEE Proceedings Software*. Special issue on Language Definitions and Tool Generation. Also in Brics RS-04-37 (BRICS, Aarhus University, Denmark).

[Maidl 2007] Maidl, A. M. (2007). A maude implementation for object-oriented action semantics. Master's thesis, Universidade Federal do Paraná. (In Portuguese).

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

[Mosses 2005] Mosses, P. D. (2005). A constructive approach to language definition. *Journal of Universal Computer Science*, 11(7):1117–1134.

[Mosses and Musicante 1994] Mosses, P. D. and Musicante, M. A. An action semantics for ML concurrency primitives. In *Proc. FME'94 (Formal Methods Europe, Symposium on Industrial Benefits of Formal Methods)*. Lect. Notes Comp. Sci. 873, Springer, Berlin (Oct 1994).

[Watt 1991] Watt, D. (1991). In *Programming Language Syntax and Semantics*. Prentice Hall International (UK).

A μ-Pascal using OOAS + Visitor Patterns

```
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:
        sort-of _ : T → Sort
        sort-of ⟦ “boolean” ⟧ = truth-value
        sort-of ⟦ “integer” ⟧ = inteter
End Class

Class Declaration
    syntax:
        Dec
    semantics:
        elaborate _ : Visitor → Action
End Class

Class Constant
    extending Declaration
    using I:Identifier, E:Expression, T:Type
    syntax:
        Dec ::= “const” I “:” T “=” E
    semantics:
        elaborate V:Visitor = visitDec V
End Class

Class Variable
    extending Declaration
    using I:Identifier, E:Expression, T:Type
    syntax:
        Dec ::= “var” I “:” T [ “=” E ]
    semantics:
        elaborate V:Visitor = visitDec V
End Class
```

```
Class VarSeq
    extending Declaration
    using $D_1$:Declaration, $D_2$:Declaration
    syntax:
        Dec ::= $D_1$ “;” $D_2$
    semantics:
        elaborate $V$:Visitor = visitDec $V$
End Class

Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Visitor $\rightarrow$ Action
End Class

Class Assign
    extending Command
    using $I$:Identifier, $E$:Expression
    syntax:
        Cmd ::= $I$ “:=” $E$
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class

Class Selection
    extending Command
    using $E$:Expression, $C_1$:Command, $C_2$:Comando
    syntax:
        Cmd ::= “if” $E$ “then” $C_1$ [ “else” $C_2$ ] “end-if”
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class

Class While
    extending Command
    using $C$:Command, $E$:Expression
    syntax:
        Cmd ::= “while” $E$ “do” $C$
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class
```

```
Class Sequence
    extending Command
    using $C_1$:Command, $C_2$:Comando
    syntax:
        Cmd $::=$ $C_1$ “;” $C_2$
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class

Class Expression
    syntax:
        Exp
    semantics:
        evaluate _ : Visitor $\rightarrow$ Action
End Class

Class Sum
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “+” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Subtraction
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “-” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Product
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “*” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class
```

```
Class Number
    extending Expression
    using $N$:Numeral
    syntax:
        Exp ::= $N$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class IdentifierExp
    extendig Expression
    using $I$:Identifier
    syntax:
        Exp ::= $I$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class True
    extending Expression
    syntax:
        Exp ::= “true”
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class False
    extending Expression
    syntax:
        Exp ::= “false”
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class LessThan
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp ::= $E_1$ “<” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class
```

```
Class Equal
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp ::= $E_1$ “=” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Visitor
    syntax:
        Vis ::= Declaration | Command | Expression
    semantics:
        visitDec _ : Declaration → Action
        visitCmd _ : Command → Action
        visitExp _ : Expression → Action
End Class

Class Interpreter
    extending Visitor
    semantics:
        visitDec ⟦ “const” $I$ “:” $T$ “=” $E$ ⟧ =
            |evaluate $E$
            then
            |bind token $I$ to the given value
        visitDec ⟦ “var” $I$ “:” $T$ ⟧ =
            |allocate a cell [ sort-of $T$ ]
            then
            |bind token $I$ to the given cell
        visitDec ⟦ “var” $I$ “:” $T$ “=” $E$ ⟧ =
            ||evaluate $E$
            |and
            ||allocate a cell [ sort-of $T$ ]
            then
            ||bind token $I$ to the given cell
            |and
            ||store the given value in the given cell
        visitDec ⟦ $D_1$ “;” $D_2$ ⟧ =
            |elaborate $D_1$
            before
            |elaborate $D_2$
        visitCmd ⟦ $I$ “:=” $E$ ⟧ =
            |evaluate $E$
            then
            |store the given value in the cell bound to $I$
```

```
visitCmd ⟦ “if” $E$ “then” $C_1$ “end-if” ⟧ =
    |evaluate $E$
    then
    ||execute $C_1$
    |else
    ||complete
visitCmd ⟦ “if” $E$ “then” $C_1$ “else” $C_2$ “end-if” ⟧ =
    |evaluate $E$
    then
    ||execute $C_1$
    |else
    ||execute $C_2$
visitCmd ⟦ “while” $E$ “do” $C$ ⟧ =
    unfolding
    ||evaluate $E$
    |then
    ||execute $C$ and then unfold else complete
visitCmd ⟦ $C_1$ “;” $C_2$ ⟧ =
    |execute $C_1$
    and then
    |execute $C_2$
visitExp ⟦ $E_1$ “+” $E_2$ ⟧ =
    ||evaluate $E_1$
    |and
    ||evaluate $E_2$
    then
    |give sum(the integer #1, the integer #2)
visitExp ⟦ $E_1$ “-” $E_2$ ⟧ =
    ||evaluate $E_1$
    |and
    ||evaluate $E_2$
    then
    |give difference(the integer #1, the integer #2)
visitExp ⟦ $E_1$ “*” $E_2$ ⟧ =
    ||evaluate $E_1$
    |and
    ||evaluate $E_2$
    then
    |give product(the integer #1, the integer #2)
visitExp ⟦ $N$ ⟧ =
    give valuation $N$
visitExp ⟦ $I$ ⟧ =
    |give the value bound to token $I$
    or
    |give the value stored in the cell bound to token $I$
```

```
          visitExp ⟦ “true” ⟧ =
              give true
          visitExp ⟦ “false” ⟧ =
              give false
          visitExp ⟦ $E_1$ “<” $E_2$ ⟧ =
              ||evaluate $E_1$
              |and
              ||evaluate $E_2$
              then
              |give less(the integer #1, the integer #2)
          visitExp ⟦ $E_1$ “=” $E_2$ ⟧ =
              ||evaluate $E_1$
              |and
              ||evaluate $E_2$
              then
              |give equal(the integer #1, the integer #2)
End Class

Class Micro-Pascal
    using $D$:Declaration, $C$:Command
    syntax:
          Prog ::= “begin” $D$ “;” $C$ “end”
    semantics:
          run _ : Prog → Action
          run ⟦ “begin” $D$ “;” $C$ “end” ⟧ =
              |elaborate $D$
              hence
              |execute $C$
End Class
```

B μ-Pascal using OOAS + LFLv2 + Visitor Patterns

```
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:
        sort-of _ : T → Sort
        sort-of ⟦ “boolean” ⟧ = truth-value
        sort-of ⟦ “integer” ⟧ = integer
End Class

Class Declaration
    syntax:
        Dec
    semantics:
        elaborate _ : Visitor → Action
End Class

Class Constant
    extending Declaration
    using I:Identifier, E:Expression, T:Type
    syntax:
        Dec ::= “const” I “:” T “=” E
    semantics:
        elaborate V:Visitor = visitDec V
End Class

Class Variable
    extending Declaration
    using I:Identifier, E:Expression, T:Type
    syntax:
        Dec ::= “var” I “:” T [ “=” E ]
    semantics:
        elaborate V:Visitor = visitDec V
End Class
```

```
Class VarSeq
    extending Declaration
    using $D_1$:Declaration, $D_2$:Declaration
    syntax:
        Dec $::=$ $D_1$ “;” $D_2$
    semantics:
        elaborate $V$:Visitor $=$ visitDec $V$
End Class

Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Visitor $\rightarrow$ Action
End Class

Class Assign
    extending Command
    using $I$:Identifier, $E$:Expression
    syntax:
        Cmd $::=$ $I$ “:=” $E$
    semantics:
        execute $V$:Visitor $=$ visitCmd $V$
End Class

Class Selection
    extending Command
    using $E$:Expression, $C_1$:Command, $C_2$:Comando
    syntax:
        Cmd $::=$ “if” $E$ “then” $C_1$ [ “else” $C_2$ ] “end-if”
    semantics:
        execute $V$:Visitor $=$ visitCmd $V$
End Class

Class While
    extending Command
    using $C$:Command, $E$:Expression
    syntax:
        Cmd $::=$ “while” $E$ “do” $C$
    semantics:
        execute $V$:Visitor $=$ visitCmd $V$
End Class
```

```
Class Sequence
    extending Command
    using $C_1$:Command, $C_2$:Comando
    syntax:
        Cmd $::=$ $C_1$ “;” $C_2$
    semantics:
        execute $V$:Visitor = visitCmd $V$
End Class

Class Expression
    syntax:
        Exp
    semantics:
        evaluate _ : Visitor $\rightarrow$ Action
End Class

Class Sum
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “+” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Subtraction
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “-” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Product
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp $::=$ $E_1$ “*” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class
```

```
Class Number
    extending Expression
    using $N$:Numeral
    syntax:
        Exp ::= $N$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class IdentifierExp
    extendig Expression
    using $I$:Identifier
    syntax:
        Exp ::= $I$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class True
    extending Expression
    syntax:
        Exp ::= “true”
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class False
    extending Expression
    syntax:
        Exp ::= “false”
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class LessThan
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp ::= $E_1$ “<” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class
```

```
Class Equal
    extending Expression
    using $E_1$:Expression, $E_2$:Expression
    syntax:
        Exp ::= $E_1$ “=” $E_2$
    semantics:
        evaluate $V$:Visitor = visitExp $V$
End Class

Class Visitor
    using: $O$:LFL
    syntax:
        Vis ::= Declaration | Command | Expression
    semantics:
        visitDec _ : Declaration $\rightarrow$ Action
        visitCmd _ : Command $\rightarrow$ Action
        visitExp _ : Expression $\rightarrow$ Action
End Class

Class Interpreter
    extending Visitor
    semantics:
        visitDec $[\![$ “const” $I$ “:” $T$ “=” $E$ $]\!]$ =
            $O$.Declaration.Paradigm.Imper.Constant.elaborate-constant(
                    $I$, sort-of $T$)
        visitDec $[\![$ “var” $I$ “:” $T$ $]\!]$ =
            $O$.Declaration.Paradigm.Imper.VarDec.elaborate-variable(
                    $I$, sort-of $T$)
        visitDec $[\![$ “var” $I$ “:” $T$ “=” $E$ $]\!]$ =
            $O$.Declaration.Paradigm.Imeper.Variable.elaborate-variable(
                    $I$, encapsulate evaluate $E$)
        visitDec $[\![$ $D_1$ “;” $D_2$ $]\!]$ =
            $O$.Declaration.Shared.VariableSequence(
                    encapsulate elaborate $D_1$,
                    encapsulate elaborate $D_2$)
        visitCmd $[\![$ $I$ “:=” $E$ $]\!]$ =
            $O$.Command.Paradigm.Imper.Assign.execute-assignment(
                    $I$, ecapsulate evaluate $E$)
        visitCmd $[\![$ “if” $E$ “then” $C_1$ “end-if” $]\!]$ =
            $O$.Command.Shared.Selection.execute-if-then(
                    encapsulate evaluate $E$, encapsulate execute $C_1$)
        visitCmd $[\![$ “if” $E$ “then” $C_1$ “else” $C_2$ “end-if” $]\!]$ =
            $O$.Command.Shared.Selection.execute-if-then-else(
                    encapsulate evaluate $E$, encapsulate execute $C_1$,
                    encapsulate execute $C_2$)
        visitCmd ⟦ “while” $E$ “do” $C$ ⟧ =
            $O$.Command.Shared.While(
                    encapsulate evaluate $E$, encapsulate execute $C$)
        visitCmd ⟦ $C_1$ “;” $C_2$ ⟧ =
            $O$.Command.Shared.Sequence.execute-sequence(
                    encapsulate execute $C_1$, encapsulate execute $C_2$)
        visitExp ⟦ $E_1$ “+” $E_2$ ⟧ =
            $O$.Expression.Shared.Sum.evaluate-sum(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
        visitExp ⟦ $E_1$ “-” $E_2$ ⟧ =
            $O$.Expression.Shared.Subtract.evaluate-sub(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
        visitExp ⟦ $E_1$ “*” $E_2$ ⟧ =
            $O$.Expression.Shared.Product.evaluate-prod(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
        visitExp ⟦ $N$ ⟧ =
            give valuation $N$
        visitExp ⟦ $I$ ⟧ =
            $O$.Expression.Shared.Identifier.evaluate-identifier($I$)
        visitExp ⟦ “true” ⟧ =
            $O$.Expression.Shared.True.evaluate-true()
        visitExp ⟦ “false” ⟧ =
            $O$.Expression.Shared.False.evaluate-false()
        visitExp ⟦ $E_1$ “<” $E_2$ ⟧ =
            $O$.Expression.Shared.LessThan.evaluate-less-than(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
        visitExp ⟦ $E_1$ “=” $E_2$ ⟧ =
            $O$.Expression.Shared.Equality.evaluate-equality(
                    encapsulate evaluate $E_1$, encapsulate evaluate $E_2$)
End Class

Class Micro-Pascal
    using $D$:Declaration, $C$:Command
    syntax:
        Prog ::= “begin” $D$ “;” $C$ “end”
    semantics:
        run _ : Prog → Action
        run ⟦ “begin” $D$ “;” $C$ “end” ⟧ =
            |elaborate $D$
            hence
            |execute $C$
End Class
```