Prototyping Action Semantics Using Functional Languages

Paulo Borba* Silvio Meira† André Santos‡

Departamento de Informática
Universidade Federal de Pernambuco

Abstract

Action Semantics is a development of Denotational Semantics used to write formal descriptions of programming language semantics.

In this work we describe an extension to a lazy functional language (Lazy ML) to support Action Semantics notation, so that one can (almost) directly translate Action Semantics specifications of programming languages into executable prototypes of their interpreters.

Lazy ML's action notation extension proved to be a very useful tool in the process of validation of specifications, as it makes possible to rapidly obtain a prototype interpreter of a language directly from its specification. We describe also how the system was used to produce a prototype interpreter for a functional language.

Keywords

Action Semantics, Semantic Prototyping, Semantics of Programming. Languages

1 Introduction

The formal description of a programming language is an essential document for its development, given its many important roles: it is a guide to the implementation, documents design decisions, establishes standards to be followed by implementations, provides the basis to prove program correctness in what is concerned to its specification, may be used as a reference manual for the language, etc. Informal descriptions are not as useful, for they are usually incomplete, inconsistent and ambiguous.

The syntax of programming languages is traditionally described in a formal way using context free grammars. For the definition (specification) of their semantics there are many formalisms: Operational [5], Axiomatic [2] and Denotational [6]. These formalisms,

*This author's current address:Department of Computing Science, Oxford University, e-mail: Paulo.Borba@prg.oxford.ac.uk

†e-mail: silvio@di.ufpe.br. Work of this author partly funded by CNPq grant 300133/85-CC, CNPq, Brazil. The other two authors are supported by CNPq research scholarships.

‡This author's current address:Department of Computing Science, Glasgow University, e-mail: andre@dcs.glasgow.ac.uk

initially used in theoretical studies, were not very successful when used to describe "real" programming languages like Pascal or ADA. In these cases the resulting descriptions were incomplete or extremely long and very hard to understand.

Action Semantics [4] was developed to make possible the formal specification of "real" programming languages, taking care of many of the problems in the formalisms described above. For that end, pragmatic aspects were considered: readability, modularity and reusability of the specification. These aspects are easily seen in the example we will show, specially if compared to other formalisms like denotational semantics.

Another very important point when developing a programming language is that of obtaining a prototype compiler or interpreter for the language as soon as possible. This is important for validating the language definition and for finding errors in the specification. Although Action Semantics descriptions are very operational and intuitive, there is no formal and direct way to transform a specification into a running prototype. This can be done in an entirely ad hoc way, where one cannot assure the correctness of the transformation process, and possibly loosing the level of abstraction one would like to have while specifying a language. In this case the effort to get the prototype will change the major issue of the work from the language itself to its implementation.

If it were possible to obtain a running interpreter directly from the Action Semantics specification of the language, it would be much easier and faster to try new characteristics of the programming language, and possibly improve them, what would be harder with a prototype developed in a different way. This way the specification would be in direct correspondence with the implementation and its changes could be immediately reflected in a new version of the interpreter. The cost of the continuous development of a prototype would be much reduced.

In this work we describe the implementation and use of an Action Semantics interpreter. It consists of a series of modules written in Lazy ML [1] implementing the Action Semantics notation. Other lazy functional languages could be used as well, as we use only purely functional constructs, that are present in most functional languages. This way, one can rapidly and with minimal effort write a specification and have it compiled and running. It has proven to be very useful as a tool for verifying the syntactic and type correctness of Action Semantics descriptions (there are no tools available for as yet), as well as a rapid way of getting an interpreter for a language from its formal definition, and to validate the specification.

A similar approach to the same problem is shown in [7] describing how to implement a subset of an action notation similar to that described in [4]. The main difference from our approach is that there it is not described how handle sorts neither support for specific semantic entities. In [8] a prototyping method is presented to solve these problems, and differs from our work basically by having a more complete sort handling mechanism. In both works the prototyping system is written in Standard ML, and uses specific characteristics of the language not available in other functional languages, like exception handling. Our approach was developed independently of both works and uses purely functional constructs.

We initially introduce Action Semantics, then describe part of its notation and show how the library of action notation functions was built, concluding by detailing a method for translating Action Semantics descriptions into executable prototypes. Then we describe how we translated the original action semantics description of a functional language to obtain directly from its specification the prototype of a running interpreter written in the action notation extension of Lazy ML.

2 Action Semantics

Action Semantics is a development of Denotational Semantics used to write formal descriptions of programming language semantics. This goal is achieved through modularity and readability of the formal descriptions.

Action Semantics is denotational (or "compositional") in that the semantics of each phrase of the language (expressions, declarations, etc.) is expressed in terms of the semantics of its subphrases.

The main difference between Action and Denotational Semantics is in the entities used to give meaning to programs. Denotational Semantics uses high order functions, expressed in lambda notation and Action Semantics uses special entities called *actions*, which have a (reasonably) simple operational interpretation and quite nice algebraic properties.

The Action Notation is used to express the basic actions to compute new values, bind tokens to values, etc., and action combinators that are used to build complex actions and denote sequential execution, action selection, etc. The generality of these combinators, which follow the suitable algebraic laws, provides the inherent modularity of the semantic descriptions. Thanks to the use of commonly used words with their intuitive meanings, actions expressed in the notation are intentionally verbose and suggestive, making it possible to gain a (broad) impression of a language's semantics from a superficial reading of its Action Semantics description.

Actions can be performed, and they process information gradually, in a step-by-step way. Therefore, they are inherently more operational than functions. When performed, an action (which may be part of an enclosing action) can:

- *complete*, corresponding to normal termination (the performance of the enclosing action proceeds normally) or
- *escape*, corresponding to exceptional termination (the enclosed action is skipped until the escape is trapped) or
- *fail*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too) or
- *diverge*, corresponding to non termination (the enclosing action also diverges).

Considering the performance of an action as corresponding to the behaviour of a program, actions can be used to represent the semantics of programs. Moreover, actions can be used to represent the contribution that parts of the program give to the semantics of the whole program. This is expressed through the mapping of programming language phrases to the actions they denote.

The information processed by actions are elements (and possibly sorts) of data. Abstractions are used to encapsulate actions and are classified as data as well, although actions themselves are not.

The various kinds of information give rise to different "facets" of the actions, according to the kind of information they process at a time:

- *control* facet, which ignores information processing.
  Ex: $\boxed{A_1 \text{ and then } A_2}$ is the action that executes A_1, and if A_1 finishes A_2 is executed. A_1 and A_2 are actions.

- *functional* facet, processing transient information (actions are given and give data). Transient information consists of simple data, possibly tagged, corresponding to intermediate results.
  Ex: $\boxed{A_1 \text{ then } A_2}$ is the action that executes A_1, and then executes A_2; A_1 receives the data the compound action receives, A_2 receives the data generated by A_1, and the data generated by the compound action are those generated by A_2.

- *declarative* facet, processing scoped information (actions receive and produce bindings). Scoped information consists of bindings of tokens to data, corresponding to symbol tables.
  Ex: $\boxed{A_1 \text{ before } A_2}$ is the action that executes A_1, and then executes A_2; A_1 receives the bindings received by the compound action. A_2 also receives the bindings produced by A_1, A_1 having precedence. The bindings produced by the compound action are the ones produced by A_1 and A_2, A_2 having the precedence.

- *imperative* facet, processing stable information (actions reserve and unreserve cells, and change the data stored in the cells). Stable information consists of data stored in cells, corresponding to values assigned to variables.
  Ex: $\boxed{\text{store } D_1 \text{ in } D_2}$ is the action that stores the datum D_1 in the cell D_2.

For space reasons, it is not possible to give a detailed account of Action Semantics here.

The semantic description style we use follows that used in [4], where the abstract syntax of the language is defined first, then the semantic entities that will express the semantic values of programs, and finally the semantic functions, expressed by actions that map the phrases of the abstract syntax to semantic values.

The notation used to specify the abstract syntax is a form of context-free grammar, similar to BNF.

The Semantic Entities module describes the semantic domains that are used, to whom the programs will be mapped, and operations over these domains (*sorts*).

In Semantic Functions we describe the meaning of a program through the mapping of its abstract syntax to semantic entities.

3 Action Semantics Notation in Lazy ML

The action notation in Lazy ML is structured as a library of predefined functions and data types defined as Lazy ML modules.

Basic data (sets, maps, etc.) are defined as abstract data types, and their respective operations (union, dom, etc.) defined as functions over them.

The basic, functional, declarative and imperative actions (combinators) are defined as high order functions that receive other actions (functions) and information about the state of transient, scoped and stable information (the state before the performance of the action) as input and give behaviours (the state after the action performance plus its status, e.g. complete, fail or escape) as result. Another possible implementation for actions could be to implement them as algebraic data type with its combinators and primitive actions as constructors. In this approach, instead of directly compiling the specification we would define a function to interpret the combinators during execution.

The state before and after action performance consists of a 4-tuple containing the information about the state of the evaluation. The 4-tuple represents the transient, scoped, stable and control information.

The transient information component can contain the three types of information that can be given to (or given by) an action from the functional facet point of view:

- Non-tagged information, meaning that the action that receives this information is given data with no tags, like in give D, where D is a datum.
- A set of tagged information, meaning that the action that receives this information is given data with tags, like give D tag[n]. The set is represented as a mapping from tags (n, represented as integers) to data (D).
- No transient information, meaning that the action receives no information.

In the scoped information field we have a mapping from tokens to data, representing the current binding information.

The stable information field contains the status of the storage (memory cells). It contains a mapping from cells (addresses) to data, or an empty mapping meaning that no cells were reserved, in case there is no storage information.

Finally the control information contains auxiliary information that is used internally in the implementation, having no equivalent in the action semantics definition.

An action behaviour contains the same kind of information as the state plus information whether it has completed, escaped or failed.

The basic semantic entities (basic data) of action notation were implemented as constructors of an algebraic data type named `datum`, mimicking the datum semantic entity of action notation. In reality, any new semantic entities that are included in a specification must be defined adding a new constructor to the ones already defined.

abstraction is the sort of data that encapsulates actions. The encapsulated action is performed when the abstraction is enacted. In the implementation it is represented by the `Abstraction` constructor, that has an `Act` (that is, a function with type `(Info -> Behaviour)`) as its component.

As an example of how Action Semantics combinators (actions) are implemented receiving the state information and giving an action behaviour, let us see how the give functional action is expressed in Lazy ML. give D is an indivisible action that completes, giving the datum D, failing when D yields 'nothing'. This action is implemented as follows:

```
give dat (info as (t,sc,st,c)) =
  let d = eval dat info
  in if d = Nothing then Fail (NoTransInfo,NoScInfo,st,NoConInfo)
                    else Complete (NonTagged d,NoScInfo,st,NoConInfo)
```

where `datum` is a dependent datum that is evaluated with the current information (`info`) yielding `d`. When `d` is `Nothing` (nothing) the action fails with the received storage information and no other information. Otherwise, the action completes with the received storage information and the transient information `NonTagged d`.

All operators are defined similarly, that is, as receiving the same parameters as in action notation plus the state information. Then, if it receives a datum, the datum is evaluated with the current information and the behaviour of the action is expressed as a `Behaviour` with the respective changes in the state information. If the action receives other actions (subactions) as parameters its evaluation depends on the evaluation of its subactions, so they are evaluated with information corresponding to the behaviour of the enclosing action. The implementation of the then functional action is an example:

```
(a1 then' a2) (i as (t,sc,st,c)) =
    case (a1 i) in
      Complete (t1,sc1,st1,c1) :
        case (a2 (t1,sc,st1,c)) in
```

```
        Complete (t2,sc2,st2,c2) :
          Complete (t2,MScInfo sc sc1 sc2,st2,MConInfo c c1 c2)
      || Escape (t2,sc2,st2,c2) : Escape (t2,sc2,st2,MConInfo c c1 c2)
      || Fail (t2,sc2,st2,c2) : Fail (t2,sc2,st2,MConInfo c c1 c2)
      end
|| any : any
end
```

It is defined as a Lazy ML function that receives two actions a1 and a2 (that is two functions, possibly the `then'` itself with two other actions as arguments), and the state information i with its four components. The action a1 is evaluated with the state information received by the `then'` combinator. If a1 fails or escapes, the whole action behaves the same way, and a2 is not even evaluated (the same happens if a1 diverges). If the evaluation (its behaviour) completes, a2 is then evaluated with the state information `(t1,sc,st1,c)`, that is, with the transient and storage information generated by a1 and the scoped and control information given to the `then'` combinator. If the evaluation of a2 completes, the behaviour of the action a1 `then'` a2 is also `Complete` with state information composed by the transient and storage information generated by a2 (t2 and st2), and scoped and control information that is a combination of those received by the `then'` combinator and those generated by a1 and a2 (`MScInfo sc sc1 sc2` and `MConInfo c c1 c2`). If a2 escapes or fails a1 `then'` a2 also escapes or fails with transient, scoped and storage information generated by a2, and a combination of control information generated by a1 and a2 and received by the `then'` combinator. If a2 diverges the whole action also diverges. The other combinators are similarly implemented.

4 From Action Semantics to an Interpreter

The method for transforming an Action Semantics description in a working interpreter prototype in Lazy ML consists, in most cases, of a simple syntactic translation. Some minor changes are needed to handle new semantic entities the description might contain in a proper way. The modules of the action semantics notation can be implemented as Lazy ML modules, to keep the implementation as modular as the specification.

Syntax. The abstract syntax is translated into a series of algebraic data types that will function as the language syntax thereafter. It is quite straightforward to derive an algebraic data type from the syntax.

Semantic Entities. The semantic entities defined as basic data in action notation are already implemented as constructors of the `datum` algebraic data type. New semantic entities that are specific to the language being defined must be included as new constructors for `datum`, if needed. If new constructors are introduced in `datum`, some modifications must also be made in the definition of functions which compare semantic entities, e.g. the equality operator must be modified to be able to compare the new entities.

Semantic Functions. The semantic functions are implemented as functions from the abstract syntax (defined as an algebraic data type) to an action. That is achieved by defining functions by pattern matching on the data types used as abstract syntax. The function body is identical to the original semantic function's body with minor syntactic changes. Let us take as an example a semantic function from the action semantics description of a subset of a lazy functional language under development in our department. We can, by comparing the definition of the semantic function **evaluate** $[\![E_1\ E_2]\!]$ in action notation and in Lazy ML's action notation. **evaluate** $[\![E_1\ E_2]\!]$ checks whether E_1 is an abstraction (in

this case the denotation of a function) and gives the result of its application to the argument E_2, or a constructor, giving the corresponding construction. It is defined in action notation as follows:

```
evaluate ⟦E1 E2⟧ =
    | evaluate ⟦E1⟧
    then before
    | | | give closure (abstraction(evaluate ⟦E2⟧)) tag [2]
    | | and then
    | | | give the resulting value tag [1]
    | then
    | | | check the resulting value[1] is an abstraction
    | | and then
    | | | enact the abstraction[1] with the argument value[2]
    | | or
    | | | check the resulting value[1] is a Value-Constructor
    | | and then
    | | | give construction(the Value-Constructor[1], the argument value[2]) ;
```

In the Lazy ML version of action notation this definition is transformed into

```
evaluate (ES e1 e2) =
    evaluate e1
  then
      ((givetag (closure (Abstraction (evaluate e2))) 2
      andthen
        givetag (the (resulting value)) 1)
    then'
        ((check ((thetagged (resulting value) 1) is' (an abstraction))
        andthen
          enact ((thetagged abstraction 1) with
                 (thetagged (argument value) 2)))
      or
         (check ((thetagged (resulting value) 1) is' (a ValueCon))
        andthen
          give (construction (thetagged ValueCon 1,
                              thetagged (argument value) 2)))))
```

Most of the semantic functions change as little as this one when redefined in Lazy ML.

5 Conclusions

The Action Semantics description of a language is an important document for its development and use, and can be very helpful in verifying correctness of implementations. It is much more readable than descriptions given in other formal methods such as denotational semantics and the specifications are inherently modular. This makes them highly reusable, because many language constructs have identical meaning in many programming languages.

The Action Semantics description of a programming language, together with a working prototype of an interpreter for the language that is obtained directly from its semantic

description, can be used to try new ideas in the design of the language and find errors much earlier in the process of its development.

Many simple (syntax and typing) errors were found in the original specification of the example language of this project when we first tried to compile it, demonstrating the usefulness of the system. They were easily found when using the system, and would be very difficult to find without it. These problems could be easily solved if there were syntax and type checkers for Action Semantics specifications.

Although the process of transforming the original Action Semantics description in one proper for compilation is not formal, the proximity of the two notations (where most definitions need minor syntactic changes) makes the process of transformation very natural and intuitive.

It should be possible to formalise the transformation step and build a syntax directed translator that will make most of the transformation, translating the original specification syntactically to the compiler notation, preserving its semantics.

Another important feature will be to support the Action Semantics communicative facet. That was not possible due to difficulties in handling processes in functional languages like Lazy ML. An implementation of the same modules in a functional language supporting processes (e.g. the functional language A), seems to be possible [3].

Acknowledgements

Many thanks are due to Professor Peter Mosses, who most kindly sent us his lecture notes on-the-fly, for a course in Recife in the fall term 1989.

Hermano Moura suggested modifications and made very interesting comments on an early draft of this paper.

References

[1] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Department of Computer Science, Chalmers University of Technology, Göteborg, draft edition, 1988.

[2] J. A. Goguen. Parameterized programming. *IEEE Trans. on Software Eng.*, pages 528–543, September 1984.

[3] S. R. L. Meira. Processes and functions. In *TAPSOFT'89*, number 351 in LNCS, Barcelona, 1989. Springer-Verlag.

[4] P. D. Mosses and D. A. Watt. The use of Action Semantics. In M. Wirsing, editor, *Conference on Formal Description of Programming Concepts III.* IFIP TC2, North-Holland, Amsterdan, 1987.

[5] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, September 1981.

[6] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[7] D. A. Watt. Executable semantic descriptions. *Software—Practice and Experience*, 16(1), January 1986.

[8] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall International, 1991.