STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 1, 2012

PRINCIPLES OF ACTION SEMANTICS FOR FUNCTIONAL PROGRAMMING LANGUAGES

VILIAM SLODIČÁK, VALERIE NOVITZKÁ

Abstract. In this paper we present a short introduction into foundations of action semantics and its application in functional paradigm. We discuss the use of action semantics for functional paradigm. The obtained results are demonstrated on the particular well-known example from informatics - the Fibonacci numbers. The computation of Fibonacci numbers has been implemented in the object-oriented functional language OCaml and the description of the program is given in action semantics.

1. Introduction

Action semantics is one of the newest methods for defining the meaning of constructions of the programming languages. Formal definition of programming language is an inseparable part of the definition of every programming language. It allows us to understand the behavior of programs by unambiguous way. Traditional methods - an operational and denotational semantics belong in the group of the most used semantical methods. The operational semantics (a semantics of small steps) plays a crucial *rôle* in implementation of the programming language, because it concerns the details of the program execution. Denotational semantics, often called also mathematical semantics, expresses the meaning of programs by functions over the mathematical structures (domains). It has been used in the design of programming languages, but its mathematical complexity is quite hard for IT experts to understand. The action semantics avoids the disadvantages of the former methods. It arises from the denotational semantics but it provides the meaning of programs by user-friendly style. It uses English phrases so the definitions of semantics of

Received by the editors: February 4, 2012.

2010 *Mathematics Subject Classification.* 68Q55.

1998 *CR Categories and Descriptors.* F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages – *Partial evaluation*; D.3.3 [**Programming languages**]: Language Constructs and Features – *Recursion*.

Key words and phrases. Action semantics, functional paradigm, actions, semantical description.

the program constructions are more readable. Action semantics is fully equivalent with other semantical methods, such as denotational semantics, natural semantics, operational semantics or axiomatic semantics [8].
Until now the action semantics has been defined for the imperative languages. But its principles are also appropriate for defining the semantics of the languages of functional paradigm. In our article, we present the principles of action semantics for the new functional object-oriented language - *OCaml* [1, 4]. The results obtained are demonstrated on the example of computation the Fibonacci numbers where we apply the results from the previous works [13, 14] about the recursion. In the section 2 we describe fundamentals of action semantics. In the next section we formulate some basic principles of action semantics in functional paradigm. The last section presents an example of applying the action semantics in functional paradigm.

2. Basic notions about Action semantics

The framework of action semantics [7] was initially developed at the University of Aarhus by Peter D. Mosses, in collaboration with David Watt from University of Glasgow. One of its main advantages over other frameworks is pragmatic: action-semantic descriptions can scale up easy to real programming languages [10, 16]. Action semantics deals with the three kinds of semantic entities: actions, yielders and data. Fundamentals of action semantics are actions which are essentially dynamic computational entities. They represent the computational behavior by using the values passed to them to generate new values that reflect changes in the state of the computation. In other words - the performance of an action directly represents the information of processing the behavior and reflects the gradual step-wise nature of computation: each step of an action performance may access and/or change the current information.

Other semantic entities used in action semantics are yielders and data. The information processed by actions consists of items of data, organized in structures that give access to the individual items. Data could contain:

- *mathematical entities* (e.g. truth values, numbers, characters, strings, lists, sets, and maps);
- *computational entities* (tokens and cells);
- *compound entities* (messages and contracts).

Yielders are another entities; they can be evaluated to yield data during action performance. The data yielded may depend on the current information:

- the given transients;
- the received bindings;

- the current state of the storage (not applied in functional programming).

The actions are main kind of entities; the yielders and data are subsidiary. The notation used for specifying actions and the subsidiary semantic entities is called action notation [7]. In action semantics, the semantics of a programming language is defined by decomposition of program phrases to actions. The performance of these actions relates closely to the execution of the program phrases. Primitive actions can store data in storage cells, bind identifiers to data, compute values, test truth values, etc. [11].

A performance of an action which may be a part of an enclosing action either:

- *completes*, corresponding to normal termination;
- *escapes*, corresponding to exceptional termination;
- *fails*, corresponding to abandoning an alternative;
- *diverges*, corresponding to deadlock.

2.1. **Action semantics facets.** The different kinds of information give rise to the so called *facets* of actions which have been classified according to [7]. They are focusing on the processing of at most one kind of information at a time:

- *the basic facet*, processing independently of information (control flows);
- *the functional facet*, processing transient information (actions are given and give data);
- *the declarative facet*, processing scoped information (actions receive and produce bindings);
- *the imperative facet*, processing stable information (actions reserve and dispose cells of storage, and change the data stored in cells);
- *the communicative facet*, processing permanent information (actions send messages, receive messages in buffers, and offer contracts to agents) [7].

2.2. **Action notation and combinators.** The standard notation for specifying actions consists of primitive actions and action combinators. Examples of action combinators are depicted on the following figures: action combinator `and` (Fig. 1), action combinator `and then` (Fig. 2), action combinator `then` (Fig. 3) and action combinator `or` (Fig. 4). In diagrams, the scoped information flows from left to the right whereas transients flow from top to the bottom.

- A_1 `and` A_2 (Fig. 1) allows the performance of the two actions to be interleaved. No control dependency in diagram, so actions can be performed collaterally.

- A_1 `and then` A_2 (Fig. 2) performs the first action and then performs the second one.
- A_1 `then` A_2 (Fig. 3) performs the first action using the transients given to the combined action and then performs the second action using the transients given by the first action. The transients given by the combined action are the transients given by the second action.
- A_1 `or` A_2 (Fig. 4) arbitrarily chooses one of the subactions and performs it with given transients and received bindings. If the chosen action fails, it perform the other subaction with original transients and bindings.

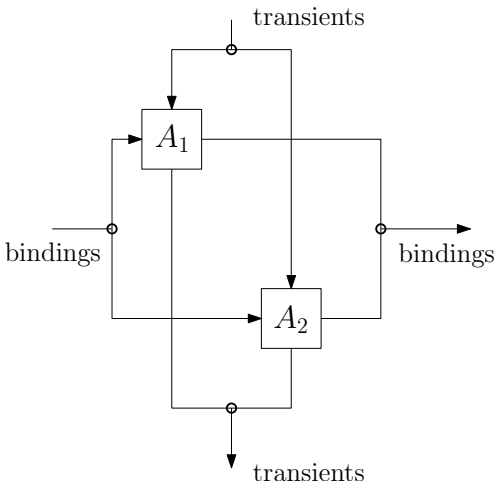

FIGURE 1. Action combinator A_1 *and* A_2

The data entities consist of mathematical values, such as integers, Boolean values, and abstract cells representing memory locations, that embody particles of information. Sorts of data used by action semantics are defined by algebraic specifications. Yielders encompass unevaluated pieces of data whose values depend on the current information incorporating the state of the computation. Yielders occur in actions and may access, but they can not change the current information.

The standard notation for specifying actions consists of primitive actions and action combinators. Action combinators combine existing actions, normally using infix notation, to control the order which subactions are performed in as well as the data flow to and from their subactions. Action combinators are used to define sequential, selective, iterative, and block structuring control flow as well as to manage the flow of information between actions. The

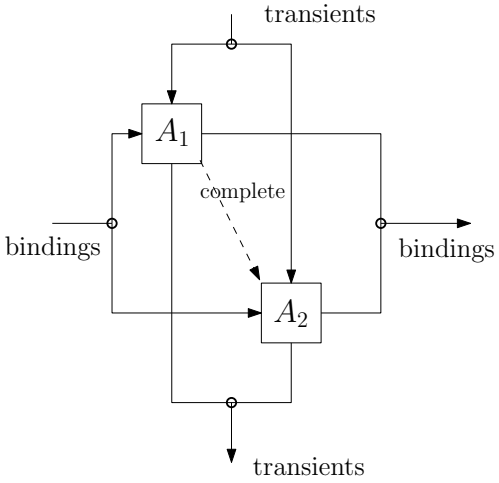

FIGURE 2. Action combinator A_1 *and then* A_2

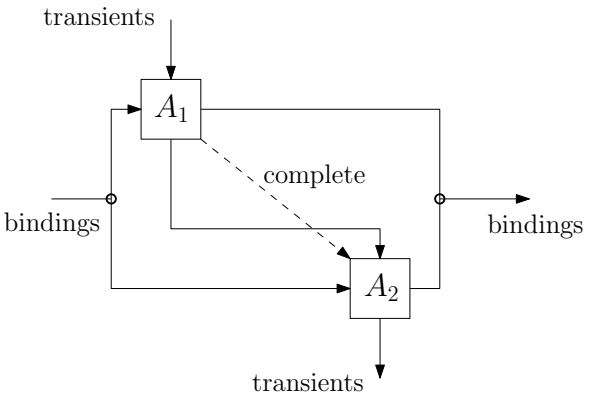

FIGURE 3. Action combinator A_1 *then* A_2

standard symbols used in action notation are ordinary English words. In fact, action notation is very near to natural language:

- terms standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g. `check it and then escape`;
- terms standing for data and yielders form noun phrases, e.g. `the items of the given list`.

These simple principles for choice of symbols provide a surprisingly grammatical fragment of English, allowing specifications of actions to be made fluently readable. The informal appearance and suggestive words of action notation

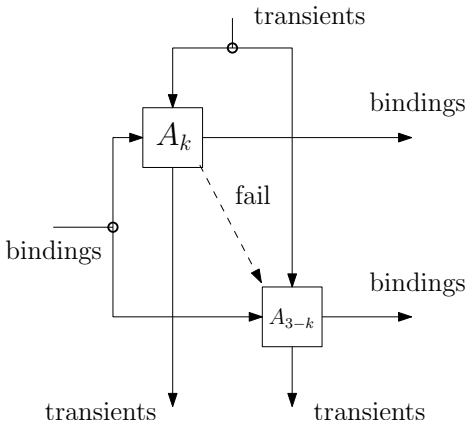

FIGURE 4. Action combinator A_1 *or* A_2 (for $k = 1$ or $k = 2$)

should encourage programmers to read it. Compared to other formalisms, such as λ-notation, action notation may appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also take into consideration that each action combinator usually corresponds to a complex pattern of applications and abstractions in λ-notation. In any case, the increased length of each symbol seems to be far outweighed by its increased perspicuity.

3. Action semantics in functional paradigm

Action semantics can be successfully used also for the description of functional programs [15]. In action semantics we use generally three main actions for the description of programming languages:

- **execute**$[\![$*statement*$]\!]$ - used for executing of statements;
- **elaborate**$[\![$*declaration*$]\!]$ - used with declarations;
- **evaluate**$[\![$*expression*$]\!]$ - used for evaluating expressions.

In functional paradigm we use only two main actions: *evaluate* and *elaborate*. Action *execute* is not used in functional paradigm; it is a standard action for imperative paradigm. Typical for functional programs is that they do not deal the storage. Therefore we will not use actions of imperative facet for allocating memory locations, storing values and getting values from cells in memory in our action semantics descriptions of functional programs.
The actions **evaluate** and **elaborate** we usually use in the form

$$\textbf{evaluate}[\![e]\!]s\,[n \mapsto val_0]\,,$$

where e is an expression to be evaluated in the input state denoted s where n has the value val_0.
Important for functional paradigm is the evaluation of the expressions and the elaboration of the functions. To allow referring them in the program code, they are associated to names (identifiers). These associations are called bindings. A binding can be *global*, when declared at the top level of the source code, or *local*, when declared in a *let* or *letrec* expressions that contain it. The difference between *let* and *letrec* expressions is that in the latter one a mutual recursion is allowed. We provide this description of evaluation of simple expression:

elaborate⟦`let` I`:Var = E:Expression`⟧ =
  evaluate ⟦ `E` ⟧
    `then bind` I `to the given value`

After declaration we are able to use it anytime in our program. The value is bound to its identifier, so we can get the value of this expression simply by using *evaluate* action:

evaluate⟦ I`:Var` ⟧ =
  `give the value bound to` I

Description of a function with one argument should seem like this:

elaborate⟦`let` I_f`:Var` I_{p1}`:Var = E:Expression`⟧ =
  evaluate⟦`E`⟧
    `then bind` I_f `to the given value`

In the expression E the parameter of the function is used. The value of the function we can get simply by action *evaluate*:

evaluate⟦ I_{p1}`:Var` ⟧ =
  `give the value bound to` I_{p1}

General definition for a function with two or more arguments:
elaborate⟦`let` I_f`:Var` $<$ I_p`:Var` $>^+$ `= E:Expression`⟧ =
  evaluate⟦`E`⟧
    `then`
  `bind` I_f `to the given value`

3.1. **Related work.** In 1997, S. B. Lassen started to develop the functional part of a theory of action semantics for reasoning about programs. Action

notation, the specification language of action semantics, was given an evaluation semantics, and operational techniques from process theory and functional programming have been applied in the development of a versatile action theory [3]. Peter D. Mosses showed in [6] the functional action notation which has extended the basic action notation and data notation with a few primitive actions, one new combinator and some notation for yielders. This notation belongs to the group of languages based on composition of functions, without exlicit mention of arguments, such as FP. But there is also another group of functional languages based on application of functions to arguments, such as Standard ML. On the other hand, David A. Watt presented in [17] an action-semantics description of Standard ML, as evidence for the claimed merits of action semantics.

4. Example

We present the description of functional program in action semantics at the well-known algorithm: Fibonacci numbers. In mathematics, the Fibonacci numbers are the numbers in the integer sequence where the first two Fibonacci numbers are 0 and 1 (sometimes first two Fibonacci numbers are considered 1 and 1), and each subsequent number is the sum of the previous two [2, 5, 12]. The construction of the sequence of Fibonacci numbers is as follows:

- $F_0 = 0$, sequence is (0);
- $F_1 = 1$, sequence is (0, 1);
- $F_2 = F_0 + F_1 = 0 + 1 = 1$, sequence is (0, 1, 1);
- $F_3 = F_1 + F_2 = 1 + 1 = 2$, sequence is (0, 1, 1, 2);
- $F_4 = F_2 + F_3 = 1 + 2 = 3$, sequence is (0, 1, 1, 2, 3);
- $F_5 = F_3 + F_4 = 2 + 3 = 5$, sequence is (0, 1, 1, 2, 3, 5);
- *etc.* ...

In mathematical terms we define the Fibonacci sequence by the linear recursive function:

$$F_n = \begin{cases} 1 & \text{if } n = 0; \\ 1 & \text{or } n = 1; \\ F_{n-1} + F_{n-2} & \text{otherwise } (n > 1). \end{cases}$$

The recursive function for the calculation of Fibonacci numbers is based on the "Divide et Impera" method. In the language *OCaml* it has the form:

```
let rec fibDnC(n) =
  if (n==0 || n==1)
       then 1
  else
       fibDnC(n-1) + fibDnC(n-2);;
```

4.1. **Description in action semantics.** We use a substitution for the term that calculates the Fibonacci number.

Let expression E be:

$$E = \texttt{if } (n == 0 \;||\; n == 1) \texttt{ then } 1 \texttt{ else } fibDnC(n-1) + fibDnC(n-2).$$

Next we elaborate the function $fibDnC(n)$

```
elaborate ⟦ let rec fibDnC (n) = E⟧ =
 recursively bind fibDnC to
  closure of
   abstraction of
    evaluate⟦E⟧ =
 recursively bind fibDnC to
  closure of
   abstraction of
    evaluate⟦n⟧
      and then
    give the TruthValue of
     (the given number is equal to the number 0
       or
      the given number is equal to the number 1)
     then
      check the given TruthValue
         and then
          give the number 1
       or
      check not the given TruthValue
         and then
          give the sum of
           (elaborate⟦fibDnC(n − 1)⟧
             and
            elaborate⟦fibDnC(n − 2)⟧)
```

The value of argument given by expression n in function $fibDnC(n)$ we get in the action **evaluate**:

```
evaluate ⟦n⟧ =
   give the value bound to n
```

where the value of the input expression is simply evaluated and returned as output value.

4.2. **The description of an example in Action semantics.** We show a partial elaboration of the function $fibDnC(n)$ in the state for the input argument $n = 5$.

elaborate $[\![fibDnC\ (n) = E]\!]s\,[n \mapsto 5] =$
 give the value bound to
  closure of
   abstraction of
    evaluate$[\![E]\!]s\,[n \mapsto 5] =$

 give the value bound to
  closure of
   abstraction of
    give the number 5
      and then
    give the $TruthValue$ of
    (the given number is equal
       to the number 0
      or
     the given number is equal
       to the number 1)
    then
     check not the given $TruthValue$
        and then
         give the sum of
          (**elaborate**$[\![fibDnC(n)]\!]s\,[n \mapsto 4]$
            and
          elaborate$[\![fibDnC(n)]\!]s\,[n \mapsto 3]$)

Next step is the elaboration of the function for the input values $n = 4$ and $n = 3$.

The elaboration of the function $fibDnC(n)$ is given by recursive evaluation of the Fibonacci numbers calculation. Here we omit some steps and show only the final step after evaluation of the given terms. Here, in the final step of description the function $fibDnC(n)$ is being called with the input value $n = 0$.

elaborate $[\![fibDnC\ (n) = E]\!]s\,[n \mapsto 0] =$
 give the value bound to
  closure of
   abstraction of
    evaluate$[\![E]\!]s\,[n \mapsto 0] =$

```
give the value bound to
  closure of
    abstraction of
     give the number 0
       and then
     give the TruthValue of
     (the given number is equal
        to the number 0
       or
      the given number is equal
        to the number 1)
     then check the given TruthValue
         and then give the number 1
```

In the following description we replaced the clause `the given number` with the concrete values by reason of showing partial results in the evaluation.

```
give the value bound to
  closure of
    abstraction of
          give the sum of
           (the number 1 and the number 1)
       and then
          give the sum of
           (the number 2 and the number 1)
       and then
          give the sum of
           (the number 3 and the number 2)
       and then
          give the sum of
           (the number 5 and the number 3) =
give the value bound to
  closure of
    abstraction of
       give the number 8
```

The result of the function $fibDnC(n)$ for the input value $n = 5$ is equal to 8.

5. Conclusion

In this article we have formulated new application area of action semantics for functional programming languages. We presented our approach on the Fibonacci numbers computation function which is traditionally used for illustration of the recursion in functional programming [9].

In the future we extend our approach to the object-oriented features of functional programming language *OCaml* and to construct a categorical form of action semantics which can be useful to observe the behavior of functional programs in coalgebraic terms.

Acknowledgement

This work has been supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10: Modelling, simulation and implementation of GPGPU-enabled architectures of high-throughput network security tools.

References

[1] Hickey, J. *Introduction to Objective Caml*. Cambridge University Press, 2008.
[2] Koubková, A., and Pavelka, J. *Introduction to theoretical informatics*. MatFyzPress, Charles University, Prague, 2005. (in Czech).
[3] Lassen, S. Action semantics reasoning about functional programs. *Journal Mathematical Structures in Computer Science Vol. 7*, Issue 5 (October 1997).
[4] Leroy, X. The objective caml system release 3.12. documentation and user's manual. Tech. rep., Institut National de Recherche en Informatique et en Automatique, 2008.
[5] Matoušek, J., and Nešetřil, J. *Kapitoly z diskrétní matematiky*. Nakladatelství Karolinum, Praha, Univerzita Karlova v Praze, 2000.
[6] Mosses, P. *Action Semantics*. Cambridge University Press, 2005.
[7] Mosses, P. D. Theory and practice of action semantics. In *In MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science* (1996), Springer-Verlag, pp. 37–61.
[8] Nielson, H. R., and Nielson, F. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., 2003.
[9] Nilsson, H., Ed. *Trends in Functional Programming*, vol. 7. Intellect Books, 2007.
[10] Planas, E., Cabot, J., and Gómez, C. Verifying action semantics specifications in uml behavioral models. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering* (Berlin, Heidelberg, 2009), CAiSE '09, Springer-Verlag, pp. 125–140.
[11] Ruei, R., and Slonneger, K. Semantic prototyping: Implementing action semantics in standard ML. The University of Iowa, 1993.
[12] Sedláček, J. *Úvod do teorie grafů*. Academia, Praha, 1981.
[13] Slodičák, V., and Macko, P. How to apply linear logic in coalgebraical approach of computing. In *Proceedings of the 22nd Central European Conference on Information and Intelligent Systems, September 21-23, 2011, Varaždin, Croatia* (2011), University of Zagreb, pp. 373–380. ISSN 1847-2001.

[14] Slodičák, V., and Macko, P. New approaches in functional programming using algebras and coalgebras. In *European Joint Conferrences on Theory and Practise of Software - ETAPS 2011* (March 2011), Workshop on Generative Technologies, Universität des Saarlandes, Saarbrücken, Germany, pp. pp. 13–23. ISBN 978-963-284-188-5.

[15] Slodičák, V., and Macko, P. Some new approaches in functional programming using algebras and coalgebras. *Electronic Notes in Theoretical Computer Science Vol. 279*, Issue 3 (2011), pp. 41–62.

[16] Stuurman, G. Action semantics applied to model driven engineering, November 2010. University of Twente.

[17] Watt, D. An action semantics of standard ml. *Lecture Notes in Computer Science Vol. 298/1988* (1988).

Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovak Republic

E-mail address: viliam.slodicak@tuke.sk, valerie.novitzka@tuke.sk