An Implementation of Action Semantics (DRAFT)

Hermano Moura*

University of Glasgow
Department of Computing Science
Glasgow G12 8QQ, Scotland
moura@dcs.glasgow.ac.uk

Abstract. Action semantics is a new formalism for specification of programming languages. Action notation is the formal language used in action semantics. We describe here an action notation interpreter. ANI, as we call it, interprets actions, which are the meaning of programs in action semantics. It also gives a very clear picture of the behaviour of actions. Used in conjunction with the "actioneer generator", ANI constitutes a way to obtain automatically an interpreter for a language. As an illustration for our ideas, we give a complete action semantic description of a small language and show how to use it to generate an interpreter using ANI and the actioneer generator.

1 Introduction

The use of formal notation in the specification of programming languages is of vital importance in the design and implementation of such systems. Rapid prototyping in the design process leads to a better knowledge of the language and permits the elimination of unsuitable properties at an early stage of the process. Also, some formal notations, like denotational semantics, suggest the implementation of an interpreter in the initial stages of the language design, which is very useful to make experiments to consolidate the language definition.

We present here an implementation of an interpreter for action notation, the formal notation used in *action semantics*, a new and promising formalism for specification of programming languages [3, 7, 4, 6]. We start with an overview of action semantics, followed by the definition of a small programming language, MICROML , where we put action semantics into action. In Section 4 we describe ANI – the action notation interpreter. The great practical utility of ANI is stressed when we put it to work in conjunction with the actioneer generator, which is described in Section 5, in order to obtain an interpreter for MICROML . Finally, some global considerations are discussed.

2 Action Semantics

Action semantics, like denotational semantics, is compositional: the semantics of each phrase is determined by the semantics of its subphrases. The difference is that

* Supported by CNPq, Brazil. On leave from Caixa Econômica Federal, Brazil.

the semantics of phrases (their denotations) are no longer higher-order functions: they are *actions*. Thus, action semantics can be regarded as denotational, where the denotations are actions.

In action semantics, the notation used for specify lexical, concrete, and abstract syntax is simply a form of context-free grammar (see, for example, the abstract syntax of microML in Appendix A). Actions are used as abstract entities to model behaviour. Semantic functions are specified by semantic equations, in a form similar to denotational semantic descriptions.

The semantics of a programming language is given in terms of a small number of action primitives and action combinators. This provides the possibility of reusing parts of previous action semantic descriptions when describing new languages, and facilitates the semantic comparison of languages. The notation used is verbose and suggestive, which improves readability of semantic descriptions. The aim of action semantics is to obtain better pragmatic qualities than those of denotational semantics, and it has been used to give semantics to a variety of programming languages.

2.1 Concepts

Action notation, the notation used in action semantic descriptions, has a simple operational interpretation, and quite nice algebraic properties [3]. The main concept in action semantics is the concept of an *action*. An action is an entity that can be performed, receiving and producing data. Actions can be combined using *action combinators*. A performance of an action (which may be part of an enclosing action) either: (*i*) *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or (*ii*) *escapes*, corresponding to exceptional termination (the enclosing action is skipped until the escape is trapped); or (*iii*) *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or (*iv*) *diverges*, corresponding to nontermination (the enclosing action also diverges).

Another important concept is the concept of *sort*. A sort is a *choice* of elements. When one sort is a subsort of another, all elements of the first sort are also elements of the second sort. Sorts that are subsorts of each other are regarded as identical. The notation treats sorts themselves as abstract entities, and allows operations and relations to be applied to any arguments whatsoever – elements, sorts, or even a mixture. In fact each element is regarded itself as a sort that classifies just one entity. So one could write:

$$\begin{gathered} 1 : 1 \\ 1 : \mathsf{integer} \\ 1 \leq \mathsf{integer} \\ \mathsf{integer} \leq \mathsf{integer} \end{gathered}$$

where, $x : s$ means x is an element of sort s and $s \leq t$ means that the sort s is a subsort of the sort t.

Sort symbols are treated as ordinary constants. The arguments and results of operations may be sorts as well as elements. Operations are total, and monotone with respect to subsort inclusion; they return the vacuous sort (nothing) to represent undefinedness. Moreover, sorts are treated as nondeterministic choices between the elements that they classify; a singleton sort is thus treated as its unique element.

Actions process elements (and sorts) of *data*. The datum yielded during the performance of an action depends on the *current information*, thus, *dependent data* consist of entities that can be evaluated to yield data. This evaluation cannot affect the current information. Data is a special case of dependent data, and always yields itself when evaluated.

The current information can be regarded as a triple formed by *transients*, *bindings* and *storage*. Transients, a mapping between labels and data, represent transient information, that is, data corresponding to intermediate results. Bindings, a mapping between tokens and data, represent scoped information, that is, bindings of tokens to data. Storage, a mapping between cells and data, represents stable information, that is, data stored in cells. Each type of information is characterized by a special standard sort, namely transients, bindings and storage. These various type of information give rise to the so-called *facets* of actions, according to the type of information the performance of an action affects: (*i*) the *control facet*, in which the performance of an action is independent of any information; (*ii*) the *functional facet*, which affects transient information (actions are *given* and *give* transients); (*iii*) the *declarative facet*, which affects scoped information (actions *receive* and *produce* bindings); (*iv*) the *imperative facet*, which affects stable information (actions *reserve* and *unreserve* cells of storage, and *change* the data stored in cells).

2.2 Action Notation

Basic actions are concerned only with flow of control. They include some action primitives as well as action combinators. "complete", for example, is an indivisible action that always completes. It corresponds to a null action. It gives no data. It produces no bindings. "unfold" is a dummy action, standing for the subaction of the enclosing unfolding. "unfolding A" is an action that performs A, but whenever the dummy action "unfold" is reached, A is performed in place of "unfold". It corresponds to iteration. "A_1 or A_2" is an action that makes an arbitrary choice between performing the subactions A_1 and A_2. It corresponds to the concept of nondeterministic choice between alternatives; If the chosen subaction fails, the other subaction is performed – except when a committing action has cut away the alternative. The performance of each subaction has the same initial information as the compound action. "A_1 and A_2" is an action that performs both its subactions, with arbitrary interleaving of their indivisible actions. It corresponds to arbitrary order of evaluation; an escape or failure causes any remaining parts of the subactions to be skipped. Both subactions are given the same data as the compound action. When the subactions *complete*, the compound action gives all the data that both subactions give (assuming the data are tagged disjointly). Both subactions receive the same bindings as the compound action. When the subactions *complete*, the compound action produces all the bindings that both subactions produce (assuming the tokens of the bindings are disjoint). "A_1 and then A_2" is an action that performs both its subactions in left-to-right order. However, A_2 is only performed if A_1 completes. It corresponds to sequential performance.

Functional actions are concerned only with the processing of transient information. The functional action "give d label $\#n$", where d is a dependent datum and n is a natural number, gives the elementary datum yielded by d, labelled with n. It completes when d yields a datum and fails when d yields nothing. The action "check d", where d is a dependent truth-value, completes when d yields true and fails when it yields false (or nothing). It is used as a guard that a condition is true and gives no transients. "A_1 then A_2" is an action that performs both its subactions in left-to-right order. However, A_2 is only performed if A_1 completes. It corresponds to functional composition. A_1 is given the same transients as the compound action. A_2 is given the transients given by A_1. When A_1 completes, the compound action gives only the transients given by A_2. Both subactions receive the same bindings as the compound action. When the subactions complete, the compound action produces all the bindings that both subactions produce (assuming the bindings are for disjoint sets of tokens). "the $S\#n$" is a functional dependent datum, which yields the datum labelled with the label n in the transients given to its evaluation, restricted to the sort S ("the S" is equivalent to "the $S\#0$"). For example, if the transients

$$\{(0 \mapsto \text{false}), (3 \mapsto 5.22), (4 \mapsto 3.1415)\}$$

are given to the action

give the truth-value

then it gives false. If it is given to

give the real#4

then this latter action gives 3.1415.

Declarative actions produce bindings between tokens (usually corresponding to identifiers) and data. "bind k to d" is an indivisible action that produces the binding of the token k to the datum d, failing if d yields "nothing". "recursively bind k to d" produces a single recursive binding of token k to datum d. "furthermore A" produces the bindings produced by A together with the bindings received by the compound action that do no conflict with those produced by A. "A_1 before A_2" is an action that performs both the subactions in left-to-right order. It corresponds to letting bindings accumulate. A_1 receives the bindings that are received by the compound action. A_2 receives also the bindings produced by A_1, the latter taking precedence. The compound action produces the bindings produced by A_1 and A_2, the latter taking precedence. (The functional facet, not important in this particular case, is as for "A_1 and A_2".) "the S bound to k" is a dependent datum that yields the datum to which the token k is bound in the current bindings, restricted to the sort S. It yields "nothing" if k is not bound.

Action notation adopts a simple model of storage: storage is organized as a collection of primitive *cells*, each of which may contain a single datum. The action "store d in c" is an indivisible committing action that stores the datum d in the cell c, failing if the cell is not reserved. "reserve d" is an indivisible committing action that reserves the cell d, storing "uninitialized" in it, failing if the cell is already reserved. "allocate S" is an action that chooses just one unreserved cell of the sort S, reserves it, and gives the cell (thus it involves both the functional facet and the imperative facet).

An *abstraction* is a special kind of data that incorporates an action. The incorporated action is performed when the abstraction is enacted. "enact d" is an action that performs the action incorporated in the abstraction yielded by evaluating d. The performance of the incorporated action is given no transients and receives no bindings. A single item of data can be supplied to an abstraction using the operation "with", and bindings by using the operation "closure".

Besides actions, action notation includes a set of familiar data sorts and operations – such as truth-values, numbers, lists, sets and maps – generally called *data notation*. Some of the operations are for use on proper sorts, not just individuals. For example, [min 10] natural is the sort of all natural numbers not less than 10.

2.3 Remarks

We did not cover the complete action notation – we presented only those parts necessary for a first idea on the issue and that help the reading of the rest of this article. (For example, we skip the communicative facet of actions: actions *send* and *receive* messages, and *subcontract* tasks to other *agents*, processing *permanent* information. There is also a well defined notation for modular action semantic descriptions.) For a complete and detailed presentation see Mosses [3] and Watt [7].

3 microML

Now it is time to put actions into our service: let us define microML! Firstly an informal description. (The complete action semantics of microML is shown in Appendix A.)

As the name suggests, microML is a small subset of the Standard ML core language [2]; it is a pure functional language with higher-order functions; like Standard ML it is strict; booleans and integers are the only data types present in microML (with some basic operations); it is supposed to be a statically typed language, with a polymorphic type inference system along the lines of the Standard ML type system.

The syntax of microML is very simple (see Appendix A). A program is a sequence of declarations. Each declaration binds an identifier to a value. We should also notice that only unary functions are present. The following example shows a microML program consisting of three declarations: the first one binds the identifier `f` to a lambda expression defining a function that expects a function as an argument and return the result of the application of this function to the integer `4`; the second one binds `g` to another lambda expression and the last one binds `result` to the result of the application of `f` to `g`.

```
val f = fn m => m 4
val g = fn n => 5 * n
val result = f g
```

The global effect of the program is the production of three bindings.

The semantic entities part of the semantic definition defines some sorts used in the semantic functions part. For example, value = truth-value | integer | function .

assures that functions are treated as ordinary values (truth-value and integer are standard sorts provided by data notation). We use the standard sort abstraction to give a semantic meaning to functions:

function = abstraction .

The semantic functions are the central part of the specification. It is worthwhile to examine some of them. A MICROML program is a sequence of declarations, where each declaration receives all the bindings produced by previous declarations and adds a new binding to the current bindings (environment). Thus, the following equations provide meaning to a MICROML program:

run ⟦ D ⟧ = elaborate D .
run ⟦ D P ⟧ = elaborate D before run P .

The semantic equations for declarations are shown below. The recursive binding declaration `val rec` allows the expression bound to an identifier to make a reference to the binding produced by the declaration itself. The "recursively bind _ to _" combinator captures this requirement.

elaborate ⟦ `val` I `=` E ⟧ = evaluate E then bind token of I to the value .
elaborate ⟦ `val rec` I_1 `= fn` I_2 `=>` E ⟧ =
    recursively bind token of I_1 to closure abstraction
        | |furthermore bind token of I_2 to the argument
        | hence
        | |evaluate E .

The semantic equations for expressions are very simple. Function definition and application are the most interesting cases. A lambda expression is evaluated to an abstraction with static bindings – an abstraction whose incorporated action is given some bindings whenever the abstraction is enacted. The bindings are those current for the valuation of closure abstraction A. Thus,

evaluate ⟦ `fn` I `=>` E ⟧ =
    give closure abstraction
        | |furthermore bind token of I to the argument
        | hence
        | |evaluate E .

To evaluate a function application $E_1(E_2)$, we firstly evaluate E_1, which must give an abstraction, and evaluate E_2 (this can give a function value as MICROML is higher-order), then we enact the abstraction with the result of the argument evaluation. The enaction corresponds to the performance of the action incorporated in the abstraction with the given transients (the argument#2). This is exactly what is *said* by the equation below.

evaluate ⟦ E_1 (E_2) ⟧ =
  | | evaluate E_1 then give the abstraction label #1
  | and
  | | evaluate E_2 then give the argument label #2
  then
  | enact (the abstraction#1 with the argument#2) .

The semantic specification for MICROML only gives its dynamic semantics. Static semantics (context-sensitive constraints) can also be given; where the action corresponding to a program actually checks the constraints, when performed. We could write the complete semantics of a program as

meaning of ⟦ P ⟧ = typecheck P and then run P .

where typecheck is the static semantic function of a MICROML program [4].

4 ANI

The action notation interpreter, which was implemented using STANDARD ML , includes the action notation abstract syntax (a STANDARD ML datatype) and an interpreting function that takes an action and interprets it giving an interpretation output. The interpretation output is basically a triple representing transients, bindings and storage produced by the action (it can also be an error message reporting some illegal condition arisen during interpretation).

4.1 Interpreting Actions

Three functions are used to interpret an action: `step`, `propagate` and `evaluate`. A *state* is formed by the action to be interpreted, together with the transients, bindings and storage to be received by that action. (The *initial state* is formed by the action to be interpreted, no transients, no bindings and an empty store). A *step* is an interpretation outcome; it is formed by an outcome status, transients, bindings, storage and a commitment flag that indicates whether the action committed or not. In Figure 1 we present a signature with the major types and functions defined in the implementation. The function `step` takes a state, interprets it and gives a step. For states containing a compound infix action, it uses the auxiliary function `propagate`, which takes two *stepped actions* composed by an infix combinator (a *stepped infix action*), composes them and gives a new step. For example, for the action "A_1 then A_2" we have the `step` and `propagateTHEN` functions shown in Figure 2. Finally, the `evaluate` function evaluates a dependent datum giving a datum. Figure 3 shows how the dependent datum "the $S\#n$" is evaluated, where `at [t, n]` is the datum to which `n` is mapped by the map `t`. Transients, bindings and storage (as well as the commitment flag) are propagated throughout the interpretation process.

4.2 Data Notation

We also implemented many of the operations present in data notation (such as sum, union, etc). Some of them were used in the definition of the interpreting function itself, for example the function `merge` in the definition of `propagateTHEN`.

```
signature Ani =
  sig
    type 'a AST
    type datum and sort and transients and bindings and storage and pos
    datatype state = State of pos AST * transients * bindings * storage
    and commitment = Uncommitted | Committed
    and step = Step of outcomeStatus * transients * bindings * storage * commitment
    and outcomeStatus = Completed | Escaped | Failed
    val <                  : sort * sort -> bool
    val &                  : sort * sort -> sort
    val individualToSort   : datum -> sort
    val sortToIndividual   : sort -> datum
    val evaluate           : pos AST -> datum -> datum -> datum -> datum
    val evaluateSort       : pos AST -> sort
    val propagateTHEN      : step -> step -> step
    val step               : state -> step
    val interp             : pos AST -> unit
  end
```

Fig. 1. A signature with ANI's major types and functions.

4.3 Observations and Problems

We have used some action notation laws to implement the interpretation of some actions. For example, the law

$$\mathsf{diverge} = \mathsf{unfolding\ unfold}$$

was directly applied to interpret the action "diverge" as follows

```
step (State (DIVERGE p,t,b,s)) = step (State (UNFOLDING (UNFOLD p,p),t,b,s))
```

A better example is the direct implementation of the action "allocate a cell" from the law

allocate $c \leq$ cell =
    indivisibly
        ||choose a [not in domain of current storage] cell
        |then
        ||reserve it and give it .

which was done using an auxiliary function that gives a cell that is not in a given set (`cell_not_in`).

The action "A_1 or A_2" is interpreted as follows. A random number is generated to determine whether A_1 or A_2 should be tried first. The other subaction is interpreted only if the first one fails and is uncommitted. This solution gives an interesting flavour of non-determinism for the "or" combinator (although it is not true non-determinism).

```
fun step (State (THEN (a1, a2, _), t, b, s)) =
    let
        val Step (r1, t1, b1, s1, c1) = step (State (a1, t, b, s))
    in
        if r1 = Completed
            then propagateTHEN (Step (r1, t1, b1, s1, c1))
                               (step (State (a2, t1, b, s1)))
        else if r1 = Escaped
                 then Step (r1, t1, EmptyBindings, s1, c1)
             else Step (r1, EmptyTransients, EmptyBindings, s1, c1)
    end

fun propagateTHEN
    (Step (Completed, t1,b1,s1,c1)) (Step (Completed, t2,b2,s2,c2))
    = Step (Completed, t2, merge [b1,b2], s2, commit c1 c2)

  | propagateTHEN
    (Step (Completed, t1,b1,s1,c1)) (Step (Escaped  , t2,b2,s2,c2))
    = Step (Escaped, t2, EmptyBindings, s2, commit c1 c2)

  | propagateTHEN
    (Step (Completed, t1,b1,s1,c1)) (Step (Failed   , t2,b2,s2,c2))
    = Step (Failed, t2, EmptyBindings, s2, commit c1 c2)

  | propagateTHEN _ _ = raise Error (AniError
    "propagateTHEN: Cannot propagate two stepped actions.")
```

Fig. 2. The interpretation of "A_1 then A_2".

The interleaving of "A_1 and A_2" is specialized to the case where the whole of A_1 is interpreted before the interpretation of A_2. Accordingly, the behaviour of "A_1 and A_2" is equivalent to "A_1 and then A_2". This can be a problem in cases where A_1 diverges and A_2 fails; in ANI such actions will diverge rather than fail (as it should do if a more genuine interleaving interpretation were used).

An abstraction is represented by a triple (A, t, b), where A is the incorporated action, t is a set of transients, and b is a set of bindings. These three fields are supplied by the "abstraction", "with", and "closure" operations respectively. The "enact" action is interpreted by interpreting the action A with transients t and bindings b.

The most difficult part of the work was the implementation of recursive bindings. The interpretation of "recursively bind k to d" is just a bind of k to a closure (d, t, b, s, r). The contents of the closure is the unevaluated datum d, transients, bindings, storage and a recursive binding respectively. This last component is just a binding of k to the closure itself, except that in this closure the recursive binding component is empty (the effect is that the recursive binding component is a pointer to the binding).

```
fun evaluate (THE (S, n, p)) t b s =
    let
        val S' = evaluate_sort S t b s
        val n' = mk_int n
        val d  = at [t, NaturalE n']
    in
        if d is_of_sort S' then d else NothingE
    end
```

Fig. 3. The evaluation of "the $S\#n$".

Non-elementary actions (sorts) are not implemented, for example, the action "give a natural", which is equivalent to "give 0 or give 1 or ...". However "choose a natural" is properly implemented, although the action will give the same natural element every time it is performed. In general, "choose d" will give an element of the sort yielded by the evaluation of d, if this is a subsort of distinguishable datum.

ANI implements only a fixed set of sorts, the standard ones that are subsort of datum. Sorts like act are not defined. User-defined sorts cannot be included in the present implementation, although in terms like "the $S\#n$", S can be any subsort of datum, ANI will simply treat it as the sort datum. Dependent data as

the (integer | truth-value)

are properly interpreted (or even "the 1"!).

5 The Actioneer Generator

Although ANI was designed to interpret arbitrary actions, it is very useful to interpret actions that are denotations of programs – we call such an action a *program action*. In order to achieve this, we use ANI in conjunction with the *actioneer generator*.

We represent the abstract syntax of a language by a STANDARD ML datatype, which should be provided by the user. Figure 4 shows a fragment of MICROML abstract syntax tree defined in such a way.

```
datatype       Program = DECL of Declaration
                       | DECLSEQ of Declaration * Program

and        Declaration = VAL of Identifier * Expression
                       | VALREC of Identifier * Identifier * Expression
```

Fig. 4. A fragment of MICROML's abstract syntax tree.

The objective of the actioneer generator is to generate a program, the *actioneer for $\mathcal{L}$*, from the action semantic definition of $\mathcal{L}$. The actioneer incorporates $\mathcal{L}$ action

semantics. The semantic function for an $\mathcal{L}$ program, say $\mathcal{P}$, present in the actioneer, when applied to the abstract syntax tree of $\mathcal{P}$, gives the program action for $\mathcal{P}$. Figure 5 shows the architecture of the whole system. To illustrate the use of Ani in conjunction with the actioneer generator we give a concrete example using microML
.

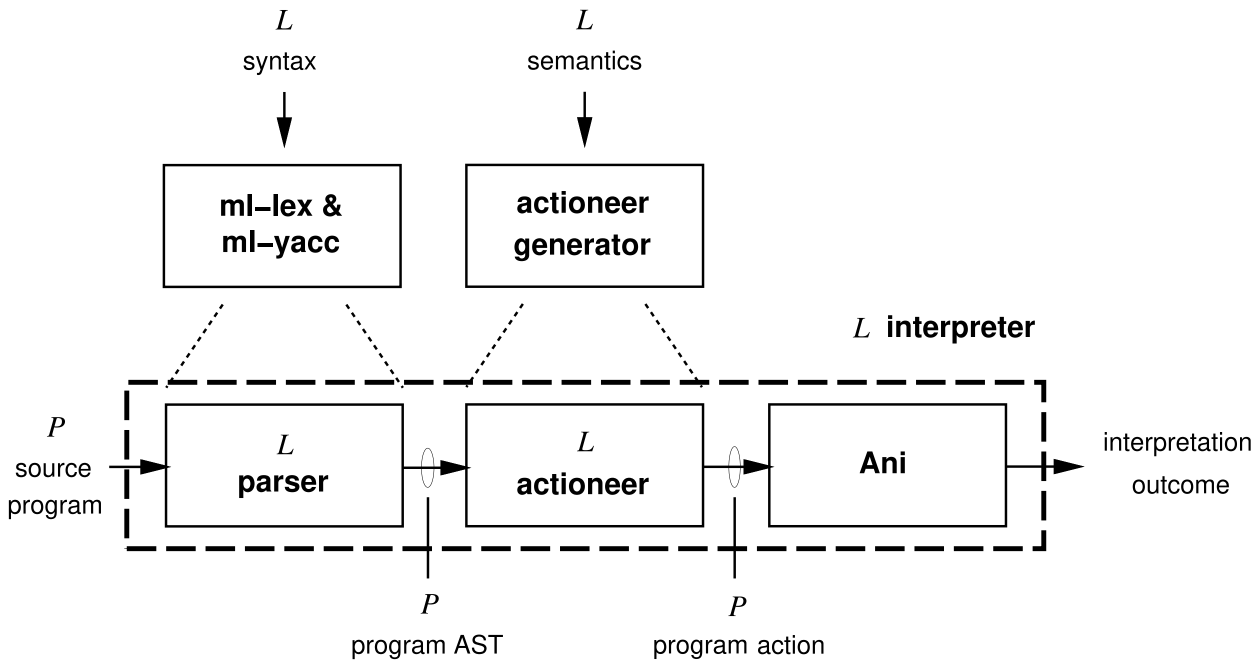

Fig. 5. Ani, the actioneer generator and an interpreter for $\mathcal{L}$.

An Example

Suppose we want to interpret the program given in Section 3. Firstly we use a parser generator to build a parser for microML (we used ML-Yacc for this purpose [5]). This parser must give an algebraic data type as the representation of a parse tree. Secondly, we obtain an actioneer for microML by application of the actioneer generator to the microML semantic definition. Figure 6 shows a piece of the actual input to the actioneer generator (compare with the semantic definition in Appendix A). Applying the parser to the source program gives the abstract syntax tree representation of the it. Now we can apply the semantic function `run`, present in the actioneer, to this tree to obtain the program action of Figure 7 (conveniently pretty printed by Ani).

```
3.2     Declarations

        introduces:     elaborate _ .
        needs:          Expressions .

(1)     elaborate _ :: Declaration -> act .

(2)     elaborate [[ VAL I:Identifier E:Expression ]] =
                evaluate E then bind tokenOf I to it .

(3)     elaborate [[ VALREC I1:Identifier I2:Identifier E:Expression ]] =
                recursively bind tokenOf I1 to closure abstraction
                        | | furthermore bind tokenOf I2 to the argument
                        | hence
                        | | evaluate E .
```

Fig. 6. The semantic equations for MICROML's declarations.

Finally, we use ANI to interpret this action. The following is the interpretation outcome:

```
Outcome        Completed!
Transients     empty-transients
Bindings       { result --> 20 }
               { g --> abstraction ... }
               { f --> abstraction ... }
Storage        empty-storage
Commitment     Uncommitted action!
```

Notice that this shows the global effect of the program, that is, the production of three bindings. Also notice the values bound to `f`, `g` and `result`. □

The implementation of the actioneer generator was very straightforward as long as we had the action notation abstract syntax definition. The present version is very simple. In a future version we intend to implement a deeper analysis of the semantic definition. At present, some consistency checks are deferred to be performed by the STANDARD ML compiler itself. Some future enhancements would include consistency checks on the semantic equations against the abstract syntax definition, sort checking, user-defined sorts and an automatically-generated TEX output (similar to the one in Appendix A).

6 Conclusion

To achieve a better understanding of action notation, to obtain action semantics executable specifications, to grasp the action notation operational semantics [3], and to have some insights on the sort checking problem, were the main initial motivations for this work.

A collection of useful functions, like a pretty printer for action trees, is available in ANI . Also, the interpreter is completely integrated to an action semantics directed compiler generator, presently being developed in Glasgow [1].

In [6, 7] Watt defines a method of executing action semantic descriptions. In his approach, actions are defined as higher-order functions (using STANDARD ML). In ANI we tried to be as closed as possible to Mosses' definitive operational semantics of action notation [3], as well as to implement all the present standard notation. Some features from the *real* action notation, like the interleaving of "and", were not implemented. For the future, however, we should try to implement these features, as well as the communicative facet of actions.

ANI seems to be a good friend to introduce to new *actioneers*. We think that, with some *healthy makeup*, it should become an interesting semantic prototyping system along the lines advocated in [6].

References

1. D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. In *Compiler Construction '92*. Springer-Verlag, oct 1992. To appear.
2. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
3. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
4. P. D. Mosses and D. A. Watt. The Use of Action Semantics. In M. Wirsing, editor, *Formal Description of Programming Concepts*, Amsterdam, 1987. North Holland.
5. D. R. Tarditi and A. W. Appel. *ML-Yacc, version 2.0*, 1990. Documentation for Release Version.
6. D. A. Watt. Executable Semantic Descriptions. *Software – Practice and Experience*, 16(1):13–43, 1986.
7. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

A The Action Semantics of MICROML

A.1 Abstract Syntax

```
(1) Program     = Declaration |
                  Declaration Program .
(2) Declaration = val Identifier = Expression |
                  val rec Identifier = fn Identifier => Expression .
(3) Expression  = true |
                  false |
                  Numeral |
                  Identifier |
                  fn Identifier => Expression |
                  Expression ( Expression ) |
                  if Expression then Expression else Expression |
                  Expression Operator Expression |
                  Operator Expression .
```

(4) Operator = not | and | or | ~ | + | - | * | / | = .

A.2 Semantic Entities

A.2.1 Data

(1) value = truth-value | integer | function .
(2) operand = truth-value | integer .
(3) operation-result = truth-value | integer .

A.2.2 Bindings

(1) token $\leq$ string .
(2) bindable(k:token) = value .

A.2.3 Abstractions

(1) function = abstraction .
(2) argument = value .
(3) function-result = value .

A.3 Semantic Functions

A.3.1 Programs

- run _ :: Program $\rightarrow$ act .

(1) run $[\![$ D $]\!]$ = elaborate D .
(2) run $[\![$ D P $]\!]$ = elaborate D before run P .

A.3.2 Declarations

- elaborate _ :: Declaration $\rightarrow$ act .

(1) elaborate $[\![$ `val` I `=` E $]\!]$ = evaluate E then bind token of I to the value .
(2) elaborate $[\![$ `val rec` I_1 `=` `fn` I_2 `=>` E $]\!]$ =
  recursively bind token of I_1 to closure abstraction
  || furthermore bind token of I_2 to the argument
  | hence
  || evaluate E .

A.3.3 Expressions

A.3.3.1 Evaluating Expressions

- evaluate _ :: Expression $\rightarrow$ act .

(1) evaluate $[\![$ `true` $]\!]$ = give true .
(2) evaluate $[\![$ `false` $]\!]$ = give false .

(3) evaluate ⟦ N ⟧ = give valuation N .
(4) evaluate ⟦ I ⟧ = give the value bound to token of I .
(5) evaluate ⟦ `fn` I `=>` E ⟧ =
give closure abstraction
||furthermore bind token of I to the argument
|hence
||evaluate E .
(6) evaluate ⟦ E_1 (E_2) ⟧ =
||evaluate E_1 then give the abstraction label #1
|and
||evaluate E_2 then give the argument label #2
then
|enact (the abstraction#1 with the argument#2) .
(7) evaluate ⟦ `if` E_1 `then` E_2 `else` E_3 ⟧ =
|evaluate E_1 then give the truth-value
then
||check (it is true) and then evaluate E_2
|or
||check (it is false) and then evaluate E_3 .
(8) evaluate ⟦ E_1 O E_2 ⟧ =
||evaluate E_1 then give the operand label #1
|and
||evaluate E_2 then give the operand label #2
then
|evaluate operator O .
(9) evaluate ⟦ O E ⟧ = evaluate E then evaluate operator O .

A.3.3.2 Evaluating Operators

- evaluate operator _ :: Operator → act .

(1) evaluate operator ⟦ not ⟧ = give not (the truth-value) .
(2) evaluate operator ⟦ and ⟧ = give conjunction (the truth-value#1, the truth-value#2) .
...
(5) evaluate operator ⟦ + ⟧ = give sum (the integer#1, the integer#2) .
...
(9) evaluate operator ⟦ = ⟧ = give (the operand#1 is the operand#2) .

A.4 Lexical Syntax

(1) Identifier = letter | Identifier letter | Identifier digit .
(2) Numeral = digit | Numeral digit .
(3) digit = `0` | `1` | ... | `9` .
(4) letter = `A` | `B` | ... | `Z` | `a` | ... | `z` .

A.4.1 Identifiers

- token of _ :: Identifier → token .

(1) token of $[\![I]\!] = I$.

A.4.2 Numerals

- valuation _ :: Numeral $\rightarrow$ integer .

(1) valuation $[\![$ 0 $]\!]$ = 0 .

...

(10) valuation $[\![$ 9 $]\!]$ = 9 .

(11) valuation $[\![N\ D]\!]$ = sum (product (valuation N, 10), valuation D) .

This article was processed using the LaTeX macro package with LLNCS style

```
||give closure (abstraction
||||furthermore bind "m" to the argument
|||hence
|||||||give the value bound to "m"
||||||then
|||||||give the abstraction label #1
|||||and
|||||||give 4
||||||then
|||||||give the argument label #2
||||then
|||||enact (the abstraction#1 with the argument#2))
|then
||bind "f" to it
before
|||give closure (abstraction
|||||furthermore bind "n" to the argument
||||hence
|||||||give 5 then give the operand label #1
||||||and
||||||||give the value bound to "n"
|||||||then
||||||||give the operand label #2
|||||then
||||||give product(the integer#1,the integer#2))
||then
|||bind "g" to it
|before
||||||give the value bound to "f"
|||||then
||||||give the abstraction label #1
||||and
||||||give the value bound to "g"
|||||then
||||||give the argument label #2
|||then
||||enact (the abstraction#1 with the argument#2)
||then
|||bind "result" to it
```

Fig. 7. The program action for the example program.