Action Transformation: An Application of Sort Inference

Kent D. Lee

May 30, 1998

Abstract

Action Semantics is a formal method of defining programming language semantics in which actions describe the manipulation of three entities: transients, bindings, and the store. Due to the high-level nature of Action Semantics, actions cannot be directly translated into efficient code in an Action Semantics-based compiler. However, by applying sort inference to an action, it is possible to transform it to a *low-level* action that can be translated into efficient code. This paper briefly introduces Action Semantics. The problem of Action Semantics-directed compilation is then demonstrated through an example. Finally, sort inference is applied to an action and it is subsequently transformed to an action more suitable for compilation.

1 Introduction

Action Semantics is a formal method of defining programming languages developed by Peter Mosses[9, 10] in collaboration with David Watt[15]. Action Semantics is based on Denotational Semantics. But Action Semantics, unlike Denotational Semantics, is designed to be both modular and readable by average programmers. A modular language specification is one that can be extended to introduce new language constructs with little or no rewriting of the existing description[10].

Another semantic method with similarities to Denotational Semantics is Monadic Semantics. Monadic Semantics exhibits modularity like Action Semantics. Monad Transformers allow the language designer to develop modular language descriptions that, through the use of partial evaluation, may be used to develop both modular interpreters and compilers[6, 7, 8]. However, Monadic descriptions are not suitable as language descriptions for the average programmer since an in depth knowledge of category theory is a prerequisite[10].

Compiler generation based on Action Semantics suffers from many of the same problems as compilers based on Monad Transformers. Actions, the computational entity of Action Semantics, are very high-level and not easily translated into a low-level form like assembly language. One approach to this problem is to make the code generator very complex and perform optimizations on the target program to simplify it. This is the approach taken in [13]. However, through the use of sort inference, an action may be simplified to a form that resembles assembly language[12, 11]. The same effect has been achieved through the use of partial evaluation[3, 4] and by the definition of a special low-level compilation semantics in the context of Monad Transformers[6].

This paper describes the problem of translating an action to a form suitable to be used in code generation in a compiler. In particular, this paper focuses on code generation for statically bound languages. Section 1.1 contains a brief introduction to Action Semantics. The reader familiar with Action Semantics can skim this section while paying careful attention to the description of combinators. For a more thorough introduction see[14]. Mosses has also written a definitive text on Action Semantics[10]. Section 2 presents an Action Semantic description and describes a compiler generator called Genesis based on Action Semantics. Section 3 describes the sort inference algorithm used in Genesis. Section 4 shows how the sort inference information may be used to eliminate transients and bindings from a program's action. Section 5 concludes and describes areas of future research in the Genesis project.

1.1 Action Semantics

In Action Semantics the action is the basic unit of computation. An action manipulates one or more of three entities: transients, bindings, and the store. Transients are intermediate results of computation. Bindings are the usual definition, a mapping of identifiers to values or locations within the store. The store is a mapping of locations, called cells, to values. In the language of Action Semantics, actions give transients, produce bindings, and store data.

Transients, bindings, and storage are all ways of organizing data. Action Semantics uses a sort system instead of a type system. Sorts are partially ordered by the subsort operator (i.e. $\leq$) with the special sort nothing as the least sort. In a sort system there is no distinction between an individual and the sort containing just that individual. For example, 6 and the sort of 6 are the same thing. Datum is the sort of all individual sorts. Data is another name for the sort of tuples of datum.

In this paper integers and cells are used, but other sorts are defined in Action Semantics and more can be added. Standard Action Semantics includes sorts such as truth-value, real, character, and map. Unlike standard Action Semantics, the Genesis compiler generator qualifies cells (a map from natural to storable) according to the datum they may contain. So instead of a cell, Genesis calls a generic cell a [datum]cell. A cell that must hold an integer is called an [integer]cell. This distinction is important to efficiently generate code for a given action.

Actions may give transients. For example,

    allocate a cell

is an action that dynamically allocates a cell and gives it as a transient. The action

    bind "M" to 5

is another action that produces a new binding of M to 5. Actions can also store values as in

    store 6 in the given cell

This action stores 6 in the cell that's given to the action. Actions not only give transients, produce bindings, and store values, but their performance can depend on the given transients, produced bindings, or stored values. In other words, an action's performance may depend on incoming transients, bindings, and storage. Every action creates some outgoing transients and bindings and may affect the contents of the store.

Actions that depend on the incoming transients, bindings, and storage contain yielders. The action

    store 6 in the given cell

contains the yielder given S which examines the incoming transients and yields the datum, d, contained in it such that $d \leq S$. To access the n^{th} element of the incoming transients the yielder given $S\#n$ may be used. Other yielders allow actions to access bindings and the store. The yielder S bound to id yields the d that's bound to id in the incoming bindings provided $d \leq S$. The yielder S stored in c yields d provided $d \leq S$ and d is stored in cell c.

Actions and yielders which use transients or give transients are said to be part of the *functional* facet of Action Semantics. Actions and yielders which produce or receive bindings are said to be part of the *declarative* facet of Action Semantics. Finally, actions and yielders which store or depend on the contents of the store are said to be part of the *imperative* facet of Action Semantics.

For actions to be interesting, we must be able to combine them into complex or compound actions. Combinators serve as constructors for complex actions. While storage is single threaded through actions (assuming no non-determinism, which isn't relevant in relation to compiler generation), transients and bindings may be propagated from one action to another in several different ways. For example, the action

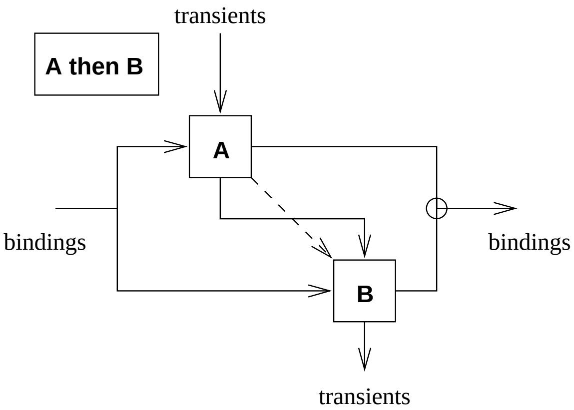

Figure 1: then combinator diagram

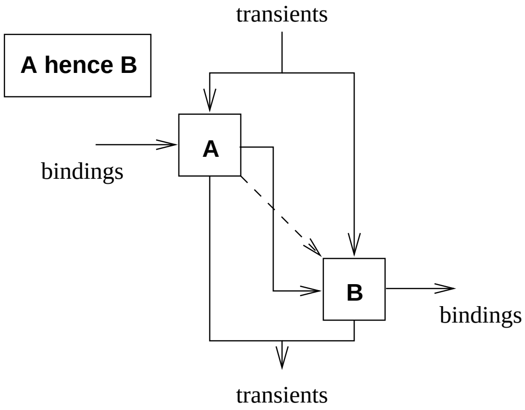

Figure 2: hence combinator diagram

| allocate a cell
then
| bind "M" to the given cell

contains the then combinator. A then B is an action that passes the outgoing transients of subaction A to the incoming transients of subaction B. The outgoing transients of B are the outgoing transients of the combined action. The incoming transients to A then B become the incoming transients to A.

To better understand combinators it is helpful to graphically see how different combinators propagate transients and bindings. Ken Slonneger [14] uses combinator diagrams to depict the nature of com-

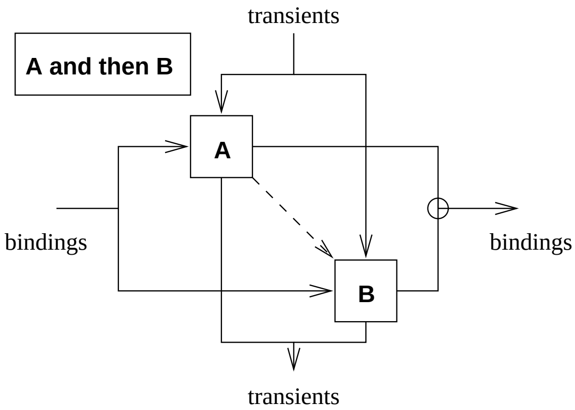

Figure 3: and then combinator diagram

binators. For instance, the then combinator is graphically depicted in figure 1. The bindings and transients are propagated as shown. The dashed line indicates that subaction A is performed before subaction B (i.e. stores performed in A will be visible to B). Some other combinators of interest are hence and and then depicted in Figures 2 and 3.

The action complete is the action that produces no bindings and gives no transients. It is of interest because it is a unit for and then and is used in algebraic identities involving the combinators then and hence.

2 A Semantic Description

To illustrate sort inference it's useful to have an example language. The calculator language will serve this purpose. The calculator is the language of expressions over integers with the addition of a memory location and the ability to store a value in it and recall its value. The language is defined via an Action Semantic description which is given in the appendix.

Action Semantic descriptions are analogous to Denotational descriptions. Semantic equations map syntax to semantics. However, the Genesis compiler generator disposes of abstract syntax and maps concrete syntactic phrases directly to their action phrases. The structure of Genesis is shown in figure 4. It generates a compiler that consists of a front end that compiles a source program to its action, and

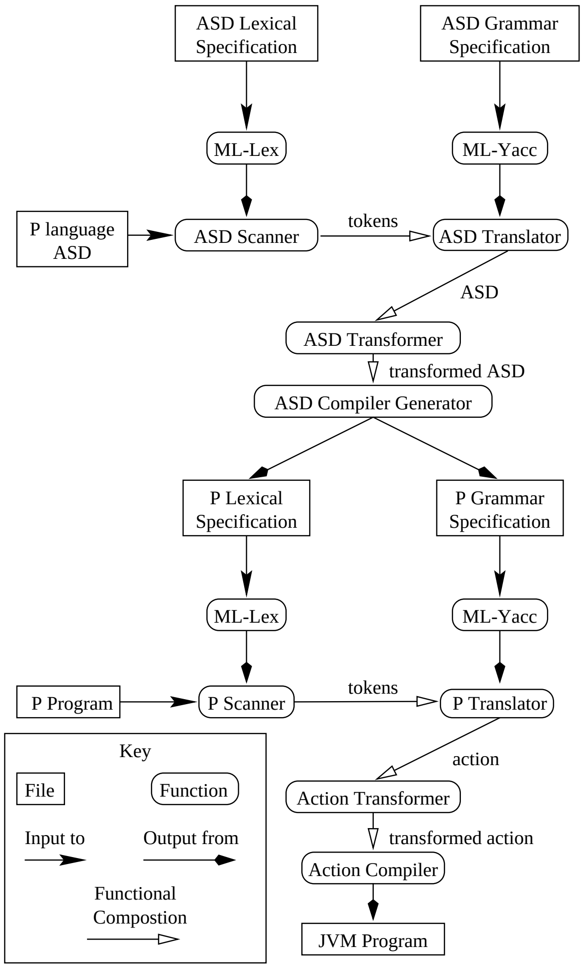

Figure 4: Genesis compiler generator

a back end that compiles an action to a target program. Initially the target program will be a Java assembly language program. However, the compiler generator can be retargeted easily by writing a back end for other target languages.

The front end generated by Genesis depends on the source language. It consists of a parser based on the concrete syntax derived from the Action Semantic description of the language. The parser translates a source program to its action. The back end is the same for every Genesis generated compiler. It consists of an Action Transformer that performs binding and transient elimination on the program action. The Action Transformer is responsible for the action transformations described in section 4.

The calculator semantics begins by allocating a cell and binding it to the identifier M. After transforming actions of the calculator language, the M binding will be eliminated and the residual action will consist of *low-level* actions that can be easily translated into assembly language. However, before transforming an action, it is necessary and useful to perform sort inference over the action. Sort inference over actions is the topic of the next section.

3 Sort Inference

Performing sort inference on actions means examining the incoming and outgoing transients and bindings to verify that they are consistent according with the meaning of the action combinators used in the action. An action's sort is the collection of these four sorts: the sorts of the incoming and outgoing transients and bindings for the action. The imperative facet is not considered in sort inference since the domain of the store cannot generally be known statically.

Each of the incoming and outgoing transients and bindings are described by a record. Transients are represented as a record that maps natural numbers to sorts. For bindings, the record maps identifiers to sorts. Sort inference on actions was first described by Even and Schmidt in [5] and was extended by Brown and Watt in [1, 2]. Watt, Brown, and Moura do not implement transients as a tuple in their compiler generator, called Actress. As a result they introduce a dialect of Action Semantics that treats transients as a binding of natural numbers to datum. The net result of this is that their dialect of Action Semantics is less modular than the original definition. This paper extends the sort inference algorithm of [5] and [2] to properly treat transients as tuples of datum.

Consider the action

| allocate a cell
then
| bind "M" to the given cell

After applying sort inference, this action has sort

$$(\{\},\{\}) \rightarrow (\{\},\{M : [datum]cell\})$$

The action takes no incoming transients or bindings and produces one binding.

Record sorts are extended to record sort schemes for the purpose of sort inference. Record sort schemes consist of a (possible) row variable and a collection of fields that map the record domain to its range. Field schemes may either be absent, present and represented as a sort scheme, or undetermined (represented as a field variable) to be resolved to present or absent later. Finally, sort schemes are either a subsort of datum or represented as a sort variable. For instance, assume the subaction

bind "M" to the given cell

is annotated with the sort scheme

$$(\{1 : \theta_1\}\gamma_3, \{\}\gamma_4) \rightarrow (\{\},\{\})$$

where the constraint $\theta_1 \& [datum]cell \neq nothing$ holds. γ_3 and γ_4 are row variables and θ_1 is a sort variable. Assume the action

allocate a cell

is annotated with the sort scheme

$$(\{\}\gamma_1, \{\}\gamma_2) \rightarrow (\{1 : [datum]cell\},\{\})$$

To find the sort of the entire action, the outgoing transients of the allocate action must be unified with the incoming transients of the bind action. The transients passed to the entire action (i.e. the empty transients) must be unified with the transients passed to the allocate action. The outgoing transients of the entire action are the same as the outgoing transients of the bind action.

The incoming bindings of the entire action (i.e. the empty bindings) must be unified with each of the incoming bindings of the subactions. The outgoing bindings of the two subactions must be merged together. All this information is contained in the fact that the then combinator is used to combine the two subactions.

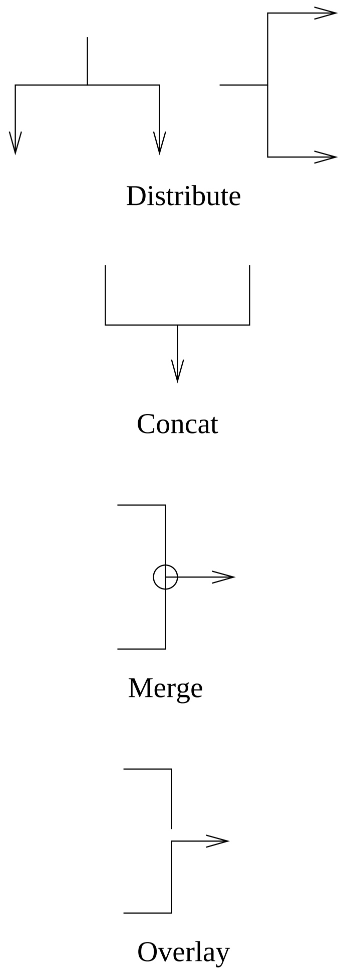

Figure 5: Derived Sort Inference Operators

Underlying the combinators used in Action Semantics are a set of derived sort inference operations which are used in the unification of record sort schemes. These derived operations are actually hinted at in the combinator diagrams of Slonneger [14]. Brown names some of these operations in [1]. The operations of interest in this paper are in figure 5. There are two other derived operations that are not described here that are needed in conjunction with the or combinator. The or combinator isn't used by the calculator language and so is omitted because of space constraints.

One derived operation is named in this paper that was avoided by Brown in [1] because of his treatment of transients. The *concat* operation is used to concatenate the outgoing transients of two subactions combined by the and then combinator. For instance, consider the actions

give 6

with sort scheme

$$(\{\}\gamma_1, \{\}\gamma_2) \rightarrow (\{1:6\}, \{\})$$

and

give 5

with sort scheme

$$(\{\}\gamma_3, \{\}\gamma_3) \rightarrow (\{1:5\}, \{\})$$

Combining these two actions with the and then combinator causes their outgoing transients to be concatenated as in

$$concat(\{1:6\}, \{1:5\}) = \{1:6, 2:5\}$$

The concatenation operator insists that the first record sort scheme be exactly known. So, $concat(\{1:6\}, \{1:5\})$ is defined whereas $concat(\{1:6\}\gamma_i, \{1:5\})$ would not be defined. This means that some polymorphic actions like

```
| give the given data
and then
| give 5
```

```
| | allocate [integer]cell
| | : ({},{}) → ({1 : θ35},{})
| then
| | bind "M" to the given cell
| | : ({1 : θ35},{}) → ({},{M : θ35})
| : ({},{}) → ({},{M : θ35})
hence
| | | | | give the 6
| | | | | : ({},{M : θ35}) → ({1 : θ6},{})
| | | | and then
| | | | | | give the 5
| | | | | | : ({},{M : θ35}) → ({1 : θ7},{})
| | | | | then
| | | | | | | store the given integer in
| | | | | | | the cell bound to "M"
| | | | | | | : ({1 : θ7},{M : θ35}) → ({},{})
| | | | | | and then
| | | | | | | give the given integer
| | | | | | | : ({1 : θ7},{M : θ35}) → ({1 : θ7},{})
| | | | | | : ({1 : θ7},{M : θ35}) → ({1 : θ7},{})
| | | | | : ({},{M : θ35}) → ({1 : θ7},{})
| | | | : ({},{M : θ35}) → ({1 : θ6, 2 : θ7},{})
| | | then
| | | | give the (sum (given integer#1,
| | | | given integer#2)) [ giving integer]
| | | | : ({1 : θ6, 2 : θ7},{M : θ35}) → ({1 : θ20},{})
| | | : ({},{M : θ35}) → ({1 : θ20},{})
| | and then
| | | give the integer stored in the cell bound to "M"
| | | : ({},{M : θ35}) → ({1 : θ33},{})
| | : ({},{M : θ35}) → ({1 : θ20, 2 : θ33},{})
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
| | : ({1 : θ20, 2 : θ33},{M : θ35}) → ({1 : θ32},{})
| : ({},{M : θ35}) → ({1 : θ32},{})
: ({},{}) → ({1 : θ32},{})
```

Figure 6: Calculator action for $(6 + 5S) * R =$

result in sort inference failure in the Genesis compiler generator. However, this type of action doesn't arise in practice and so the restriction of the derived concat operation doesn't seem to pose a problem.

Unifying record sort schemes is complex and is covered in detail in [1]. However, combining record sort schemes $A = \{fields_A\}row_A$ and $B = \{fields_B\}row_B$ can be briefly summarized as follows

1. Let $fields_{A'} = fields_B - fields_A$ and $fields_{B'} = fields_A - fields_B$.

2. Extend A and B into two new record sort schemes $A' = \{fields_A@fields_{A'}\}row_{A'}$ and $B' = \{fields_B@fields_{B'}\}row_{B'}$ whose field domains are identical. Instantiate the row variables of A and B to the newly created records (i.e. $[row_A \mapsto \{fields_{A'}\}row_{A'}, row_B \mapsto \{fields_{B'}\}row_{B'}]$)

3. Unify A' and B' according to the specified derived sort inference operation (i.e. Distribute, Concat, Merge, Overlay, Switch, or Select).

Sort inference in Genesis, like that of the Actress Compiler Generator[2], is carried out by the algorithm described above. At the end of sort inference, action sort schemes are reduced to a normalized form. A sort scheme is normalized if it is either a sort variable bound to a sort (not a sort scheme!) or a sort variable bound to a sort constructor applied to a normalized sort scheme. Normalization results in an action that is annotated with sort variables. For instance, after sort inference was performed, the calculator expression $(6 + 5S) * R =$ results in the annotated action found in figure 6 with the sort variable substitution

$$[\theta_6 \mapsto 6, \theta_7 \mapsto 5, \theta_{20} \mapsto integer,$$
$$\theta_{32} \mapsto integer, \theta_{33} \mapsto integer, \theta_{35} \mapsto [\theta_{33}]cell]$$

As might be expected the annotated action in figure 6 and the substitution given above contain valuable information that will be used in the next section.

```
| | allocate [integer]cell
| then
| | bind "M" to the given cell
hence
| | | | | complete
| | | | and then
| | | | | | complete
| | | | | then
| | | | | | | store 5 in the cell bound to "M"
| | | | | | and then
| | | | | | | complete
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in
| | | the cell bound to "M"
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 7: After eliminating θ_6 and θ_7

4 Transformation

While the action presented in figure 6 is correctly sorted, it is not suitable for efficient code generation. Fortunately, the action can be simplified by using the sort information just gathered[12]. For instance, the sort variable θ_6 is mapped to the value 6. Any yielder that yields the individual mapped by θ_6 may be replaced by 6. Similarly, any yielder that yields the datum mapped by θ_7 may be replaced by 5.

Once all dependence on θ_6 and θ_7 has been removed, all actions that give θ_6 or θ_7 may be replaced by the action complete. At this point the action has been simplified to the point shown in figure 7. This transformation is analogous to constant propagation found in most conventional compilers.

Since figure 7 includes complex actions involving complete it can be further simplified. The action complete is a unit for the and then combinator. In addition complete then $A = A$ and complete hence $A = A$ are both algebraic identities.

Applying the unit for and then and the algebraic identity involving then the action is further simplified

```
| | allocate [integer]cell
| then
| | bind "M" to the given cell
hence
| | | | store 5 in the cell bound to "M"
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in
| | | the cell bound to "M"
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 8: Simplification of subactions involving complete

```
| bind "M" to [integer]cell_0
hence
| | | | store 5 in [integer]cell_0
| | | then
| | | | give the (sum (6, 5)) [ giving integer]
| | and then
| | | give the integer stored in [integer]cell_0
| then
| | give the (product (given integer#1,
| | given integer#2)) [ giving integer]
```

Figure 9: Reduced by static allocation of the memory cell

to that shown in figure 8.

The subaction allocate a cell doesn't occur in a dynamic context (i.e. allocation is not within a loop). Therefore, the cell can be allocated statically. By extending Action Semantics to allow cells to be named, we can replace all references to the allocated cell with a named cell. We'll designate the cell's name as [integer]cell_0 and replace all yielders that yield the cell with the named cell. Then any actions that give the cell may be replaced by complete. After simplifying subactions involving complete the action is further reduced to the one found in figure 9.

Notice there are no bound to yielders left in the action. If there were, the source language and program would not be statically bound! Since there are no bound to yielders left, any occurrences of bind to actions may be replaced by complete and the appropriate identities may be used to reduce the action once again. In this example that results in the action found in figure 10.

```
| | | store 5 in [integer]cell_0
| | then
| | | give the (sum (6, 5)) [ giving integer]
| and then
| | give the integer stored in [integer]cell_0
then
| give the (product (given integer#1,
| given integer#2)) [ giving integer]
```

Figure 10: After binding elimination

The action in figure 10 is suitable for efficient code generation. The subactions resemble assembly language instructions. The transients that are left (θ_{20}, θ_{32}, and θ_{33} in the annotated action) represent the contents of the operand stack in a stack-oriented machine or the contents of symbolic registers in a register-oriented machine. Code generation is a straightforward translation of these *low-level* actions into the target language.

5 Conclusion

This paper has shown that sort inference can be applied to actions to eliminate transients and bindings from statically bound languages. Additionally, the *concat* derived sort inference operation is presented to support record sort inference over tuples. An example of action transformation is presented, demonstrating that transformed actions are, in a sense, *low-level* actions that can be directly translated into efficient code of the target language.

Future work in this area includes studying the mapping of residual transients, those transients left after transient elimination, to symbolic registers in a register-oriented machine. Another topic of research is applying sort inference to Action Semantics descriptions of language for verification and possible simplification of language descriptions.

A Calculator Action Semantics

A.1 Sorts

(1) bindable = [integer]cell

A.2 Semantic Equations

- calc _ :: Program → Act

(1) calc ⟦ E:Expr "=" ⟧ =
| | allocate an [integer]cell
| then
| | bind "M" to the given cell
hence
| evaluateExpr E .

- evaluateExpr _ :: Expr → Act

(2) evaluateExpr ⟦ E:Expr "+" T:Term ⟧ =
| | evaluateExpr E
| and then
| | evaluateTerm T
then
| give the (sum of (the given integer#1,
| the given integer#2)) [giving an integer]

(3) evaluateExpr ⟦ E:Expr "-" T:Term ⟧ =
| | evaluateExpr E
| and then
| | evaluateTerm T
then
| give the (difference of (the given integer#1,
| the given integer#2)) [giving an integer]

(4) evaluateExpr ⟦ T:Term ⟧ =
evaluateTerm T

- evaluateTerm _ :: Term → Act

(5) evaluateTerm ⟦ T:Term "*" F:Factor ⟧ =
| | evaluateTerm T
| and then
| | evaluateFactor F
then
| give the (product of (the given integer#1,
| the given integer#2)) [giving an integer]

(6) evaluateTerm ⟦ T:Term "/" F:Factor ⟧ =
| | evaluateTerm T
| and then
| | evaluateFactor F
then
| give the (quotient of (the given integer#1,
| the given integer#2)) [giving an integer]

(7) evaluateTerm ⟦ F:Factor ⟧ =
evaluateFactor F

- evaluateFactor _ :: Factor → Act

(8) evaluateFactor ⟦ G:Storable "S" ⟧ =
| evaluateStorable G
then
| | store the given integer
| | in the cell bound to "M"
| and then
| | give the given integer

(9) evaluateFactor ⟦ G:Storable ⟧ =
evaluateStorable G

- evaluateStorable _ :: Storable → Act

(10) evaluateStorable ⟦ "R" ⟧ =
give the integer
stored in the cell bound to "M"

(11) evaluateStorable ⟦ N:Integer ⟧ =
give N

(12) evaluateStorable ⟦ "(" E:Expr ")" ⟧ =
evaluateExpr E

References

[1] D.F. Brown. *Sort Inference in Action Semantic Specifications*. PhD thesis, Department of Computer Science, University of Glasgow, 1994.

[2] D.F. Brown and D.A. Watt. Sort inference in the actress compiler generator. In *Proceedings of the First International Workshop on Action Semantics*, Edinburgh, Scotland, 1994. BRICS.

[3] O. Danvy. Type-directed partial evaluation. In *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[4] O. Danvy and R. Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In *Eighth International Symposium on Programming Language Implementation and Logic Programming*, pages 182–497, 1996.

[5] S. Even and D.A. Schmidt. Type inference for action semantics. In *ESOP '90, 3rd European Symposium on Programming, volume 432 of Lecture Notes in Computer Science*, pages 118–133, Berlin, Germany, 1990. Springer-Verlag.

[6] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society, 1998.

[7] S. Liang. A modular semantics for compiler generation. Technical Report TR-1067, Yale University Department of Computer Science, 1995.

[8] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1995.

[9] Peter Mosses. An introduction to action semantics. Technical Report DAIMI PB-370, Aarhus University, Copenhagen, Denmark, 1991.

[10] Peter Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26*. Cambridge University Press, 1992.

[11] H. Moura and D.A. Watt. Action transformations in the actress compiler generator. In *Compiler Construction - 5th International Conference CC'94*, 1994.

[12] Hermano Moura. *Action Notation Transformations*. PhD thesis, Department of Computer Science, University of Glasgow, 1993.

[13] Peter Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Computer Science Department, Aarhus University, Denmark, September 1993. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[14] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Adisson Wesley Publishing Company, Inc., New York, NY, 1995.

[15] David Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1991.