Bootstrapped Semantics-Directed Compiler Generation

Stephan Diehl

FB 14 - Informatik, Universität des Saarlandes,
Postfach 15 11 50, 66041 Saarbrücken, GERMANY
diehl@cs.uni-sb.de, http://www.cs.uni-sb.de/~diehl

to appear in Proc. of 2nd International Workshop on Action Semantics Amsterdam, The Netherlands, 1999

Abstract. We introduce our natural semantics-directed generator 2BIG for compilers and abstract machines. It applies a sequence of transformations to a set of natural semantics rules including a pass separation transformation. Then we discuss how it can be used to generate a compiler and abstract machine for action notation. With the help of these components we can then generate compilers for other source languages whose semantics has been specified in Action Notation. We also briefly discuss the concept of an abstract machine language language based on the abstract machine generated for action notation.

1 Introduction

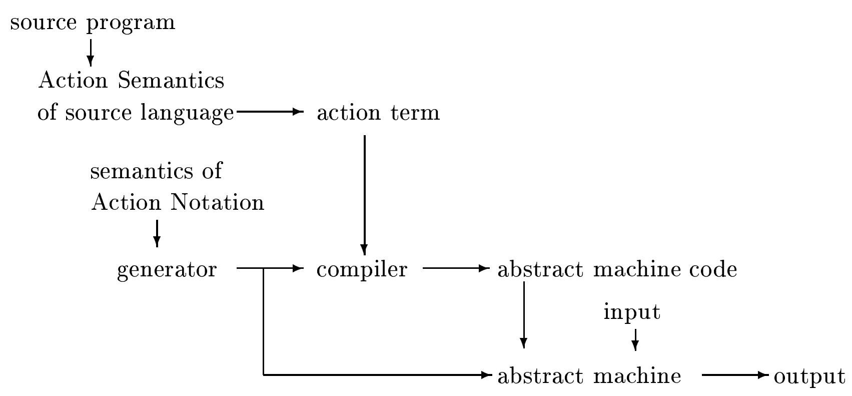
Given a semantics specification of a source language, current semantics-directed compiler generators produce compilers from the source language into a fixed target language.

$$\begin{tikzcd}
\text{semantics} \arrow[d] & \text{program} \arrow[d] & \text{input} \arrow[d] & \\
\text{generator} \arrow[r] & \text{compiler} \arrow[r] & \text{code} \arrow[r] & \text{output}
\end{tikzcd}$$

Rather than just generating compilers which translate source programs into a fixed target language, our system generates both a compiler and an abstract machine. The generated compiler translates source programs into code for the abstract machine.

$$\begin{tikzcd}
\text{semantics} \arrow[d] & \text{program} \arrow[d] & & \\
\text{generator} \arrow[r] \arrow[ddrr] & \text{compiler} \arrow[r] & \text{abstract machine code} \arrow[dd] & \\
& & & \text{input} \arrow[dl] \\
& & \text{abstract machine} \arrow[r] & \text{output}
\end{tikzcd}$$

We chose Action Notation[8] as an example of a realistic programming language, because it offers a rich set of primitives underlying both imperative and functional programming languages. Since Action Notation is used to write Action Semantics specifications, we can then combine the generated compiler for Action Notation with an Action Semantics specification of a programming language. As a result, we get a compiler from the programming language to the generated abstract machine language for Action Notation.



Thus, in general, we can bootstrap semantics-directed compiler generators (SDCG): Given an implementation of an SDCG for a semantics formalism F_1, we can get an SDCG for an semantics formalism F_2, if we have a specification of F_2 in F_1.

2 Action Semantics

Action semantics [8] has been developed to allow for useful semantics descriptions of realistic programming languages. The language used to write such semantics descriptions is called *action notation*.

The semantic entities of action semantics are *actions*, *data* and *yielders*. Actions are computational entities, they reflect the step-wise execution of programs. Data are mathematical entities like numbers, truth-values, lists and sets. Finally yielders represent unevaluated data. If the action containing a yielder is performed, the yielder evaluates to a concrete datum. Actions can become data by encapsulating them in *abstractions*, which can be *enacted* into actions again. The performance of an action may *complete* (i.e., normal termination), *escape* (i.e., exceptional termination which may be trapped), *fail* (i.e., abandoning the performance of an action which can lead to the performance of an alternative action) or *diverge* (i.e., nontermination). Actions process different kinds of information and can be classified according to which *facet* they belong: *basic* (control-flow, no data are changed), *functional* (transient information, i.e., intermediate results), *declarative* (scoped information, i.e., bindings), *imperative* (stable information, i.e., the store), *communicative* (permanent information, i.e., messages send between actions), *directive* (finite representation of self-referential bindings). Actions which process information in more than one facet are called *hybrid.* An action may *commit* and discard alternatives, e.g., in an action A_1 or A_2 representing the nondeterministic choice between two sub-actions, A_1 may commit and thus A_2 is discarded. Compound actions can be build from primitive actions using a special kind of actions called *action combinators.*

Since action semantics provides so many actions and yielders we refrain from giving an exhaustive listing but instead look at some examples.

Similar to denotational semantics the action semantics of a programming language is given by semantic equations[1]:

-
```
execute[[ X ":=" E ]] = |evaluate E
                        then
                        |store the given value in the cell bound to X
```

evaluate is a semantic function defined by semantics equations similar to the way the semantic function execute is defined here. For a concrete value of E the function evaluate yields a compound action. The action combinator A_1 then A_2 propagates the transients given to the whole action to A_1, the transients given by A_1 are propagated to A_2, and only the transients given by A_2 are given by the whole action. Thus then represents the left-to-right sequencing in the functional facet. The primitive, imperative action store Y_1 in Y_2 stores the datum produced by the yielder Y_1 in the store cell (a special kind of data) produced by the yielder Y_2. Note, that items of data are a special case of yielders, and always yield themselves when evaluated. In the above example the variable name associated with X in a concrete program would be such a special yielder.

-
```
execute[[ "while" E "do" C "od" ]] = unfolding
                                     ||evaluate E
                                     |then
                                     |||execute C then unfold
                                     ||else
                                     |||complete
```

The action combinator unfolding A performs the action A, but whenever it reaches the dummy action unfold it performs A instead. The action complete simply completes and is thus a neutral action with respect to some action combinators. The action combinator A_1 else A_2 is actually syntactic sugar for a compound action:
```
||check the given truth-value and then $A_1$
|or
||check no the given truth-value and then $A_2$
```

The action check Y completes if the yielder Y evaluates to true and fails if it evaluates to false. The yielder not Y evaluates to true (false) if Y evaluates to false (true). The yielder the given D evaluates to a transient datum of sort D given by a preceding action. There can be more than one transient datum, which is taken care of by a labeling mechanism. The action which gives a datum

[1] Instead of using parentheses to indicate precedence of actions, in action semantics we use the convention that vertical lines group actions and their arguments.

can label it, e.g., give Y label $\#n$ and later a yielder can access it, e.g., the given D label $\#n$. Now give Y is short for give Y label $\#0$ and the given D or just the D is short for the given D label $\#0$.

3 The 2BIG Generator

The 2BIG generator [2, 4] applies a sequence of established techniques to a Natural Semantics specification in order to split it into a compiler and an abstract machine. We believe that our framework, by virtue of being compositional, can be extended over time to include even more powerful analysis and transformation methods. Actually, the transformations are mostly source-to-source and after every transformation we have an executable specification again. Of these transformations pass separation is the most important one. Let p be a program and x and y the static and dynamic input to this program, then partial evaluation of p with respect to x yields a residual program p_x, such that $p_x(y) = p(x,y)$. In contrast pass separation transforms the program p into two programs p_1 and p_2 such that $p_2(p_1(x), y) = p(x, y)$. Note that here p_1 produces some intermediate data, which are input to p_2. When it comes to the generation of compiler/executor pairs, pass separation provides an immediate solution, we pass separate the interpreter *interp* into an executor *exec* and a compiler *comp*, such that: $interp(prog, data) = exec(comp(prog), data)$. Despite this potential for compiler generation there is only little work on pass separation [7, 6, 3].

Our generator first transforms the 2BIG rules into a term rewriting system:

For this, it first removes side conditions by converting them into transitions, thus there are now only transitions as preconditions. Then it factorizes rules which have a common initial sequence of preconditions. Factorization replaces these rules with a single rule which has the common initial sequence as its preconditions and for each original rule a rule is generated with its remaining preconditions. Next the generator adds a stack to the state in the transitions and stores temporary variables, i.e. variables which are not used in an intermediate transition. Variables which do not occur in the conclusion of a rule are eliminated. The last step before the actual transformation into a term rewriting system is called sequentialization. It converts all preconditions of a rule such that the result state of one transition is the start state of the next. These rules can now be easily turned into rewrite rules. Rules of the form $\frac{c_1 \vdash e_1 \rightarrow e'_1 \quad \ldots \quad c_n \vdash e_n \rightarrow e'_n}{c \vdash e \rightarrow e'}$ are converted into $\langle (c; p), e\rangle \rightarrow \langle (c_1; \ldots; c_n; p), e_1\rangle$ where p is a new variable name.

Now the resulting term rewriting system is in a form, such that pass separation can be applied which yields two term rewriting systems: one representing a compiler and one representing the abstract machine. These term rewriting systems are then further optimized to reduce the number and complexity of the abstract machine instructions, e.g. the number of arguments.

4 Transforming a 2BIG specification of Action Notation

In his PhD thesis [10] deMoura gives a natural semantics specification of a subset of action notation used in the compiler generator Actress [9]. In this specification the order of rules is important. We converted these rules into 2BIG rules adding additional preconditions, when necessary, to make the rules determinate. Then we used our system to generate a compiler and abstract machine represented as term rewriting systems.

In Section 4.1 we demonstrate the generation process by transforming the 2BIG rules for of the GIVE action.

Our specification consists of 100 2BIG rules defining the semantics of 39 action notation constructs including the control, functional, declarative and imperative facets but, as in other Action Semantics directed compiler generators, neither the communicative facet, nondeterminism nor the interleaving of actions. After transformation of side conditions we got 135 rules. Factorization resulted in 191 rules. After sequentialization we got 276 rules. Finally pass separation yielded 216 compiler rules and 276 abstract machine rules. We tested this compiler and abstract machine by translating Mini-Δ programs (e.g., Fibonacci numbers) based on an action semantics specification of the language Mini-Δ [9] into action terms. Then we compiled these action terms using the generated compiler into an abstract machine program and executed the latter by the above abstract machine rules. In other words we use a 2BIG semantics-based compiler generator to generate a compiler and abstract machine for action notation. The generated compiler is then inserted as the back end into a compiler generator based on action semantics (see Figure 1). The front end of this compiler generator was previously developed and used with a positive supercompiler for Prolog[2] as its back end [1].

4.1 Transforming the GIVE Action

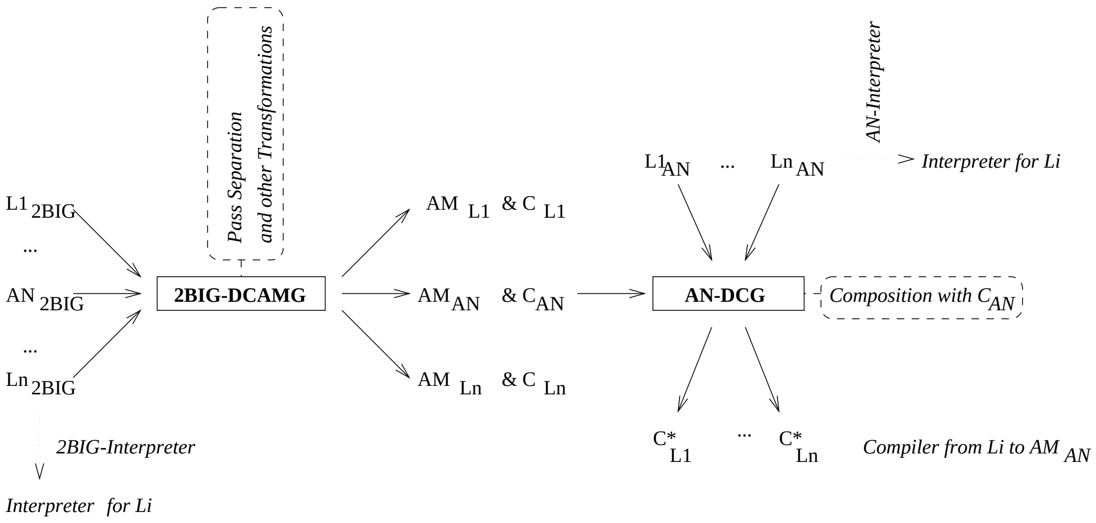
As an example we will now demonstrate step by step how our system generated the abstract machine instructions for the GIVE action.

In the 2BIG specification the following rules define the action *give* which evaluates the yielder Y and returns the resulting value D as a transient. In the rules, states are composed of the transients T, the bindings B and a single-threaded store S. Furthermore there is the outcome status O, which can be *failed* or *completed.*

$$\frac{Y \vdash [T,B,S] \rightarrow datum(D) \qquad D \neq nothing}{give(Y,N) \vdash [T,B,S] \rightarrow [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \vdash [T,B,S] \rightarrow datum(D) \qquad not(D \neq nothing)}{give(Y,N) \vdash [T,B,S] \rightarrow [failed,[],[],S]}$$

[2] Positive supercompilation [12, 11] is a program specialization technique developed in the functional community. Its adaption to Prolog is not much different from partial evaluation of Prolog [5].



Li_{2BIG}	2BIG specification of language Li
AN_{2BIG}	2BIG specification of action notation
$2BIG - DCAMG$	2BIG directed generator of compilers and abstract machines
AM_{Li}	abstract machine for language Li
AM_{AN}	abstract machine for action notation
C_{Li}	compiler from language Li into AM_{Li}
C_{Li}	compiler from action notation into AM_{Li}
Li_{AN}	action semantics specification of language Li
C^*_{Li}	compiler from language Li into AM_{AN}
$AN - DCG$	action semantics directed compiler generator

Fig. 1. Action-Semantics Directed Compiler Generation

There are two side conditions in the above rules, one is the negation of the other. Transforming the side conditions yields:

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to true}{give(Y,N) \vdash [T,B,S] \to [completed,[N \mapsto datum(D)],[],S]}$$

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to false}{give(Y,N) \vdash [T,B,S] \to [failed,[],[],S]}$$

$test_1 \vdash [D] \to D \neq nothing$

After factorization of the above rules we have:

$$\frac{Y \vdash [T,B,S] \to datum(D) \qquad test_1 \vdash [D] \to R \qquad fact_{give}(N) \vdash [[D,S],R] \to E}{give(Y,N) \vdash [T,B,S] \to E}$$

$fact_{give}(N) \vdash [[D,S],true] \to [completed,[N \mapsto datum(D)],[],S]$

$fact_{give}(N) \vdash [[D,S],false] \to [failed,[],[],S]$

$test_1 \vdash [D] \to D \neq nothing$

Now the stack (Z) is introduced and temporary variables are allocated:

$$\frac{\begin{array}{l} Y \vdash [[S|Z],[T,B,S]] \to [[S|Z],datum(D)] \\ test_1 \vdash [[[S,D]|Z],[D]] \to [[[S,D]|Z],R] \qquad fact_{give}(N) \vdash [Z,[[D,S],R]] \to [Z,E] \end{array}}{give(Y,N) \vdash [Z,[T,B,S]] \to [Z,E]}$$

$fact_{give}(N) \vdash [Z,[[D,S],true]] \to [Z,[completed,[N \mapsto datum(D)],[],S]]$

$fact_{give}(N) \vdash [Z,[[D,S],false]] \to [Z,[failed,[],[],S]]$

$test_1 \vdash [Z,[D]] \to [Z,D \neq nothing]$

Next these rules can be sequentialized:

$$\frac{\begin{array}{ll} Y \vdash [[S|Z],[T,B,S]] \to [[S|Z],datum(D)] & \\ conv_5 \vdash [[S|Z],datum(D)] \to [[[S,D]|Z],[D]] & test_1 \vdash [[[S,D]|Z],[D]] \to [[[S,D]|Z],R] \\ conv_6 \vdash [[[S,D]|Z],R] \to [Z,[[D,S],R]] & fact_{give}(N) \vdash [Z,[[D,S],R]] \to [Z,E] \end{array}}{give(Y,N) \vdash [Z,[T,B,S]] \to [Z,E]}$$

$fact_{give}(N) \vdash [Z,[[D,S],true]] \to [Z,[completed,[N \mapsto datum(D)],[],S]]$

$fact_{give}(N) \vdash [Z,[[D,S],false]] \to [Z,[failed,[],[],S]]$

$test_1 \vdash [Z,[D]] \to [Z,D \neq nothing]$

$conv_2 \vdash [[S|Z],datum(D)] \to [[[S,D]|Z],[D]]$

$conv_3 \vdash [[[S,D]|Z],R] \to [Z,[[D,S],R]]$

Now a term rewriting system is generated:

$$\begin{array}{ll}
\langle give(Y,N);C,[Z,[T,B,S]]\rangle & \\
\qquad \Longrightarrow \langle Y;conv_2;test_1;conv_3;fact_{give}(N);C,[[[S]|Z],[T,B,S]]\rangle & \\
\langle fact_{give}(N);C,[Z,[[D,S],true]] & \Longrightarrow \langle C,[Z,[completed,[N\mapsto datum(D)],[],S]]\rangle \\
\langle fact_{give}(N);C,[Z,[[D,S],false]] & \Longrightarrow \langle C,[Z,[failed,[],[],S]]\rangle \\
\langle test_1;C,[Z,[D]] & \Longrightarrow \langle C,[Z,D\neq nothing]\rangle \\
\langle conv_2;C,[[[S]|Z],datum(D)]\rangle & \Longrightarrow \langle C,[[[S,D]|Z],[D]]\rangle \\
\langle conv_3;C,[[[S,D]|Z],R]\rangle & \Longrightarrow \langle C,[Z,[[D,S],R]]\rangle
\end{array}$$

Finally we apply the pass separation transformation and we get the following compiler rules:

$$\begin{array}{ll}
give(Y,N) & \Longrightarrow \overline{give}(Y,N);Y;conv_2;test_1;conv_3;fact_{give}(N) \\
fact_{give}(N) & \Longrightarrow \overline{fact}_{give}(N) \\
test_1 & \Longrightarrow \overline{test}_1 \\
conv_2 & \Longrightarrow \overline{conv}_2 \\
conv_3 & \Longrightarrow \overline{conv}_3
\end{array}$$

And the following abstract machine rules:

$$\begin{array}{ll}
\langle \overline{give}(Y,N);C,[Z,[T,B,S]]\rangle & \Longrightarrow \langle C,[[[S]|Z],[T,B,S]]\rangle \\
\langle \overline{fact}_{give}(N);C,[Z,[[D,S],true]]\rangle & \Longrightarrow \langle C,[Z,[completed,[N\mapsto datum(D)],[],S]]\rangle \\
\langle \overline{fact}_{give}(N);C,[Z,[[D,S],false]]\rangle & \Longrightarrow \langle C,[Z,[failed,[],[],S]]\rangle \\
\langle \overline{test}_1;C,[Z,[D]]\rangle & \Longrightarrow \langle C,[Z,D\neq nothing]\rangle \\
\langle \overline{conv}_2;C,[[[S]|Z],datum(D)]\rangle & \Longrightarrow \langle C,[[[S,D]|Z],[D]]\rangle \\
\langle \overline{conv}_3;C,[[[S,D]|Z],R]\rangle & \Longrightarrow \langle C,[Z,[[D,S],R]]\rangle
\end{array}$$

5 Prototyping Tools

In Figure 1 we show how the different generators and interpreters can be used for both rapid prototyping of language specifications and generation of compilers and abstract machines. First we can use a 2BIG-interpreter to test a 2BIG-specification Li_{2BIG} of programming languages Li. Then we can generate an abstract machine AM_{Li} for the language Li and a compiler C_{Li} from Li to AM_{Li} using our 2BIG-semantics directed compiler and abstract machine generator (2BIG-DCAMG). The generator's central transformation is pass separation of term rewriting rules. In addition it applies many pre- and post-processing transformations including several optimizations.

Based on a 2BIG-specification AN_{2BIG} of a certain language, namely action notation, we generate a compiler and abstract machine for action notation. Now an action semantics specification Li_{AN} of a programming language Li can be tested both by using an action notation interpreter or by composing the action notation specification,i.e., semantics equations mapping Li programs to action notation terms, with the compiler C_{AN}. This composition results in an action semantics-directed compiler generator (AN-DCG).

Our prototyping environment includes several tools written in Prolog:

– a 2BIG interpreter
– an action notation interpreter (actually we have a handwritten interpreter, but we can also use the 2BIG interpreter to execute action terms using the 2BIG specification of action notation)
– an interpreter for compiler and abstract machine rules
– a compiler of source language programs to C using the compiler and abstract machine rules
– a compiler of compiler rules and abstract machine rules to SML

6 An Abstract Machine Language Language

Since Action Semantics is a formal language to define programming languages, we expect, that the abstract machine language AM_{AN} generated for Action Semantics is suitable to define abstract machine languages. Rather than just composing the AM_{AN} and the semantics equation which gives us AN-DCG, we could try a method similar to the combinator based approach of Wand [13, 14]. Given an Action Semantics specification of a programming language L:

1. Translate the right hand sides of the semantics equation using the generated compiler into AM_{AN} (this results in an AN-DCG).
2. Look for recurring patterns in the translated right hand side.
3. Define new instructions based on these patterns. These new instructions form an abstract machine specific for L.
4. Fold the patterns in the semantics equations by the new instructions. The resulting equations constitute a compiler into the abstract machine for L.

7 Action Semantics-Directed Compiler Generation

Now we will show how our action semantics-based compiler generator works by means of a simple example. The semantics of the language Mini-Δ is given by equations like the following one:

```
• execute⟦ X ":=" E ⟧ =
        |evaluate E
        then store the value in the cell bound to X
```

These semantic equations define a translation function from source language programs to action terms. Using this action semantics specification of Mini-Δ the following program

```
let const i=1;
    var   x:integer;
in x:=2+i end
```

is translated into the following action term

```
• |furthermore
  |||give num(1) then bind i to the given value
  ||before
  |||allocate a cell of type integer then bind x to the cell
  hence
  ||||give num(2) then give the value label#1
  |||and
  ||||||give the value stored in the cell bound to i
  |||||or
  ||||||give the value bound to i
  ||||then
  |||||give the value label #2
  ||then
  |||give add(the value #1,the value #2)
  |then
  ||store the given value in the cell bound to x
```

In our system we use prefix notation instead of the mixfix notation usually used for action terms. Thus A_1 **then** A_2 becomes **then**(A_1, A_2). The above action term in prefix notation is:

```
hence(
  furthermore(
      before(then(give(num(1),0),bind(i,the(value,0))),
             then(allocate(cell(integer)),bind(x,the(cell,0))))),
   then(
     then(
       and(then(give(num(2),0),give(the(value,0),1)),
           then(or(give(stored(value,bound(cell,i)),0),
                   give(bound(value,i),0)),
                give(the(value,0),2))),
       give(add(the(value,1),the(value,2)),0)),
     store(the(value,0),bound(cell,x))))
```

Now this action term is converted into a very long abstract machine program by the generated compiler. One reason for the length of the abstract machine program is that recurring subprograms are not shared.

$$
\begin{array}{l}
\overline{hence}(\\
\quad \overline{furthermore}(\\
\quad\quad \overline{before}(\\
\quad\quad\quad \overline{then}(\\
\quad\quad\quad\quad (\overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \overline{num}(1); \\
\quad\quad\quad\quad \overline{conv}_2; \\
\quad\quad\quad\quad \overline{test}_1; \\
\quad\quad\quad\quad \overline{conv}_3; \\
\quad\quad\quad\quad \overline{fact}_{give}(0)), \\
\quad\quad\quad\quad (\overline{bind}(i, \overline{the}(value, 0)); \\
\quad\quad\quad\quad \overline{the}(value, 0); \\
\quad\quad\quad\quad \overline{conv}_1 4; \\
\quad\quad\quad\quad \overline{test}_5; \\
\quad\quad\quad\quad \overline{conv}_1 5; \\
\quad\quad\quad\quad \overline{fact}_{bind}(i))); \\
\quad\quad\quad\quad \overline{give}(\overline{num}(1), 0); \\
\quad\quad\quad\quad \ldots
\end{array}
$$

The execution of the above program by the abstract machine in the empty environment yields the expected result: a memory cell is allocated for the variable x and the value 3 is stored in it.

In the above example the action term could be simplified before translating it into the abstract machine language. As an example

give num(2) then give the value label#1

can be simplified to give num(2) label#1. Analyses and simplifications of action terms have been investigated in de Moura's PhD thesis [10]. It would be interesting to use the simplified action terms produced by his Actress system and translate those into the generated abstract machine language.

8 Experimental Results for Optimizations

For the action notation specification, the optimizations of the generated term rewriting systems lead to a significant reduction of the number of rules both of the compiler and the abstract machine. First, by self-application, the number of compiler rules was reduced from 216 to 43. Second, using the other optimizations we got 181 instead of 276 abstract machine rules.

$$give(Y, N) \Longrightarrow \overline{give}(Y, N); Y; conv_2; test_1; conv_3; fact_{give}(N)$$

$$give(Y, N) \Longrightarrow \overline{give}; Y; \overline{comb}; \overline{fact}_{disp}(factor_{give}, N)$$

Comparing the original and the optimized compiler rule for the GIVE action we find that the following optimizations have been applied:

– The arguments to the abstract machine instruction $\overline{give}$ have been removed.
– There are no more compiler rules for $conv_2$, $test_1$, etc.
– The sequence of instructions $\overline{conv}_2;\overline{test}_1;\overline{conv}_3$ has been combined into the instruction $\overline{comb}$.
– Some abstract machine rules of $factor_{give}$ have been conflicting with rules of other instructions and thus these term rewriting rules have been factorized. This lead to the introduction of the new instruction $\overline{fact}_{disp}$.

9 Conclusion

So far our prototyping tools have been used to implement a considerable subset of Action Semantics. Instead one could also try to implement subsets of Action Semantics restricted to a few facets. As an example, to specify functional languages we could implement a version of Action Semantics without the imperative facet. As a consequence the generated abstract machine would not have a store as a compenent of its state. Another approach would be to implement an annotated version of Action Semantics and a preprocessing phase, e.g., a binding-time analysis, which translates action terms into annotated terms. Finally, rather than just experimenting with existing semantics formalism, our system can also be used to design and implement new semantics formalisms.

References

[1] Stephan Diehl. A Prolog Positive Supercompiler. in Proceedings of "Arbeitstagung Programmiersprachen", Herbert Kuchen, editor, published as Arbeitsbericht No. 58 of the Institut für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, 1997

[2] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines.* PhD thesis, University Saarbrücken, Germany, 1996. http://www.cs.uni-sb.de/~diehl/phd.html.

[3] Stephan Diehl. Transformations of Evolving Algebras. In *Proceedings of VIII International Conference on Logic and Computer Science LIRA'97*, pages 43–50, Novi Sad,Yugoslavia, 1997.

[4] Stephan Diehl. Natural Semantics Directed Generation of Compilers and Abstract Machines. *Formal Aspects of Computing (to appear)*, 1999.

[5] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *PLILP'94*. 1994.

[6] John Hannan. Operational Semantics-Directed Compilers and Machine Architectures. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.

[7] U. Jørring and W.L. Scherlis. Compilers and Staging Transformations. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[8] P.D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[9] H. Moura and D. A. Watt. Action Transformations in the ACTRESS Compiler Generator. In *CC'94*, volume LNCS 768. Springer Verlag, 1994.

[10] Hermano Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, 1993.

[11] M.H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation and GPC. In D. Sannella, editor, *Programming Languages and Systems*, volume LNCS 788. Springer Verlag, 1994.

[12] V.F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS*, 8(3), 1986.

[13] Mitchell Wand. Semantics-Directed Machine Architecture. In *Proc. of POPL'82*. 1982.

[14] Mitchell Wand. From Interpreter to Compiler: A Representational Derivation. In H. Ganzinger, N.D. Jones, editor, *Programs as Data Objects*, volume LNCS 217. Springer Verlag, 1986.