References

[1] F. daSilva. Towards a Formal Framework for Evaluation of Operational Semantics. LFCS Report ECS-LFCS-90-126, Edinburgh University, 1990.

[2] H.P. de Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, 1993.

[3] Stephan Diehl. Prolog and Typed Feature Structures: A Compiler for Parallel Computers. Master's thesis, Worcester Polytechnic Institute, Worcester, Massachusetts, 1993.

[4] Stephan Diehl. A Prolog Positive Supercompiler. 1994.

[5] John Hannan. Making Abstract Machines Less Abstract . In *Proc. of FPCA'91, LNCS 523*, pages 618–635. 1991.

[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 *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96, 1986.

[8] G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume LNCS 247, pages 22–39. Springer Verlag, 1987.

[9] P. Kursawe. How to invent a Prolog machine. In *Proc. Third International Conference on Logic Programming*, pages 134–148. Springer LNCS 225, 1986.

[10] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[11] Stephen McKeever. A Framework for Generating Compilers from Natural Semantics Specifications . In P.D. Mosses, editor, *Proc. of the 1st Workshop on Action Semantics*, BRICS-NS-94-1. University Of Aarhus, Denmark, 1994.

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

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

[14] F. Nielson and H.R. Nielson. Code Generation from Two-Level Denotational Meta-Languages. Springer LNCS 217, 1986.

[15] Ulf Nilsson. Towards a Methodology for the Design of Abstract Machines for Logic Programming. *Journal of Logic Programming*, pages 163–188, 1993.

[16] Mads Tofte. *Compiler Generators - What they can do, what they might do, and what they will probably never do.*, volume 19. Springer, EATCS Monographs in Theoretical Computer Science, 1990.

term rewriting systems are interpreted in Prolog. The next step will be to generate from the term rewriting rules an efficient implementation of the compiler and the abstract machine preferably in C. In addition one should introduce a program store and labels at this stage [5].

6 Conclusions

We presented a system, that generates a compiler and abstract machine from a 2BIG specification of a programming language. We gave the transformations used in our system and as an example we transformed the 2BIG specification of the OR action combinator and the GIVE action. We pointed out, that the system was used to generate a compiler and abstract machine for action notation and that these have been used as a backend of an action semantics-based compiler generator. Finally we discussed future work.

Acknowledgements
This work has been supported by a grant of the "Deutsche Forschungsgemeinschaft". Finally I want to thank Stephen McKeever for discussions and his comments on a draft of this paper.

is translated into the following action term:

```
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

```
hence_0(
   furthermore_0(
         before_0(
            then_0(
               ( give_0(num_0(1),0);
                 num_0(1);
                 conv_7_0;
                 test_2_0;
                 conv_8_0;
                 factor_7_0(0)),
               ( bind_0(i,the_0(value,0));
                 the_0(value,0);
                 conv_14_0;
                 test_5_0;
                 conv_15_0;
                 factor_13_0(i)));
            give_0(num_0(1),0);
            ...
```

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.

5 Future Work

Given a determinate 2BIG specification our system automatically generates a compiler and abstract machine represented as term rewriting systems. Admittedly the transformations introduce a lot of abstract machine instructions. Especially the number of conversion instructions introduced by sequentialization should be reduced by replacing similar instructions by one more general instruction, e.g. the instructions $conv_1$ and $conv_2$ defined by the following rules $conv_1, [true|R] \rightarrow R$ and $conv_2, [false|R] \rightarrow R$ might be replaced by $pop, [F|R] \rightarrow R$, but this instruction would also pop values different from $true$ and $false$. Furthermore there are instructions, which do nothing besides pattern matching, i.e. test whether the current state has a required form , and it might be safe to remove them in case, we know that the state will always have the required form. Thus these and other optimizations have to be investigated further. Currently the 2BIG specifications and the

Figure 6: Transforming the GIVE action (part III)

Now a term rewriting system is generated:

$< give(Y,N);C,[Z,[T,B,S]] > \quad \Longrightarrow \quad < Y;conv_2;test_1;conv_3;fact_{give}(N);C,[[[S]|Z],[T,B,S]] >$

$< fact_{give}(N);C,[Z,[[D,S],true]] \quad \Longrightarrow \quad [Z,[completed,[N \mapsto datum(D)],[],S]] >$

$< fact_{give}(N);C,[Z,[[D,S],false]] \quad \Longrightarrow \quad [Z,[failed,[],[],S]]$

$< test_1;C,[Z,[D]] \quad \Longrightarrow \quad [Z,D \neq nothing]$

$< conv_2;C,[[[S]|Z],datum(D)] > \quad \Longrightarrow \quad < C,[[[S,D]|Z],[D]] >$

$< conv_3;C,[[[S,D]|Z],R] > \quad \Longrightarrow \quad < C,[Z,[[D,S],R]] >$

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

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

$fact_{give}(N) \quad \Longrightarrow \quad \overline{fact}_{give}(N)$

$test_1 \quad \Longrightarrow \quad \overline{test}_1$

$conv_2 \quad \Longrightarrow \quad \overline{conv}_2$

$conv_3 \quad \Longrightarrow \quad \overline{conv}_3$

And the following abstract machine rules:

$< \overline{give}(Y,N);C,[Z,[T,B,S]] > \quad \Longrightarrow \quad < C,[[[S]|Z],[T,B,S]] >$

$< \overline{fact}_{give}(N);C,[Z,[[D,S],true]] > \quad \Longrightarrow \quad < C,[Z,[completed,[N \mapsto datum(D)],[],S]] >$

$< \overline{fact}_{give}(N);C,[Z,[[D,S],false]] > \quad \Longrightarrow \quad < C,[Z,[failed,[],[],S]] >$

$< \overline{test}_1;C,[Z,[D]] > \quad \Longrightarrow \quad < C,[Z,D \neq nothing] >$

$< \overline{conv}_2;C,[[[S]|Z],datum(D)] > \quad \Longrightarrow \quad < C,[[[S,D]|Z],[D]] >$

$< \overline{conv}_3;C,[[[S,D]|Z],R] > \quad \Longrightarrow \quad < C,[Z,[[D,S],R]] >$

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.
Figures 2 and 3 demonstrate the generation process by transforming the 2BIG rules for the OR action combinator. In the transformation of the 2BIG rules of the GIVE action shown in Figures 4, 5 and 6 we also deal with side conditions.
Our specification consists of 100 2BIG rules defining the semantics of 39 action notation constructs. After transformation of side conditions we got 135 rules. Factorization resulted in 191 rules. After sequentialization we got 276 rules. Finally pass separation yield 216 compiler rules and 276 abstract machine rules. We tested this compiler and abstract machine by translating mini-Δ programs based on an action semantics specification of the language mini-Δ [13] 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, then we use these as the backend in a compiler generator based on action semantics. The front end of this compiler generator was previously developed and used with a positive supercompiler as its backend [4].
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:

(1) execute⟦ X ":=" E ⟧ =
        | evaluate E
        then store the value in the cell bound to X

Using the action semantics specification of mini-Δ the following program

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

Figure 5: Transforming the GIVE action (part II)

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

$$\frac{Y,[[S|Z],[T,B,S]]\rightarrow[[S|Z],datum(D)] \quad test_1,[[[S,D]|Z],[D]]\rightarrow[[[S,D]|Z],R] \quad fact_{give}(N),[Z,[[D,S],R]]\rightarrow[Z,E]}{give(Y,N),[Z,[T,B,S]]\rightarrow[Z,E]}$$

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

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

$test_1, [Z, [D]] \rightarrow [Z, D \neq nothing]$

Next these rules can be sequentialized:

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

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

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

$test_1, [Z, [D]] \rightarrow [Z, D \neq nothing]$

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

$conv_3, [[[S, D]|Z], R] \rightarrow [Z, [[D, S], R]]$

3.7 Pass Separation

Pass separation works as described by Hannan [6]. Since it has been described there in great detail and we did not modify the transformation, we are not going to explain it here. Pass separation detects such parts of the term rewriting rule which can be rewritten independently of the environment, i.e. at compile time.
By $s\overset{R}{\Longrightarrow}t$ we mean, that t can be obtained from s via several steps using the rules in R. Basically, pass separation converts a set of rules R into two sets R_c and R_x, such that the following holds: if $c, e\overset{R}{\Longrightarrow}c', e'$ then

$$\begin{array}{c} c\overset{R_c}{\Longrightarrow}\tilde{c} \\ e\overset{R_c}{\Longrightarrow}\tilde{e} \\ c'\overset{R_c}{\Longrightarrow}\tilde{c'} \\ e'\overset{R_c}{\Longrightarrow}\tilde{e'} \\ \tilde{c}, \tilde{e}\overset{R_x}{\Longrightarrow}\tilde{c'}, \tilde{e'} \end{array}$$

Note, that also the environment is compiled, because there might be code sequences stored in the environment. This occurs for example in the semantics of higher order languages. The rules in R define an abstract interpreter, the rules in R_c a compiler and the rules in R_x an abstract executor. The rules in R_x belong to a special class of rewrite rules. Their right sides will only match the whole state, i.e. they never apply to subterms of the state. As a result they can be implemented more efficiently than ordinary rewrite rules.

4 Transforming a 2BIG specification of Action Notation

Action semantics [12] has been developed to allow useful semantics descriptions of realistic programming languages. The language used to write such semantics descriptions is called action notation. In his PhD thesis [2] deMoura gives a natural semantics specification of a subset of action notation used in the compiler generator ACTRESS [13]. In this specification the order of rules is important. We converted these rules into

Figure 4: Transforming the GIVE action (part I)

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:

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

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

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

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

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

$test_1, [D] \rightarrow D \neq nothing$

After factorization of the above rules we have:

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

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

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

$test_1, [D] \rightarrow D \neq nothing$

where for each rule of the form $p, e \rightarrow e'$, the command p has the form $a(x_1, \ldots, x_k)$, a is a new instruction symbol and $\{x_1, \ldots, x_k\} = \mathcal{V}(e') - \mathcal{V}(e)$.

3.6 Generation of Term Rewriting Systems

After sequentialization the instructions of each precondition can be proved in the environment resulting from its preceeding precondition. This property enables us to combine the instructions and generate term rewriting rules:
Rules of the form $c, e \rightarrow e'$ are transformed into the rewrite rule $(c; p), e \rightarrow p, e'$, where p is a new variable name, which will be bound to the program rest when the rule is applied.
Rules of the form

$$\frac{c_1,e_1\rightarrow e_1' \quad \ldots \quad c_n,e_n\rightarrow e_n'}{c,e\rightarrow e'}$$

are converted into

$$(c; p), e \rightarrow (c_1; \ldots; c_n; p), e_1$$

where p is a new variable name.

Figure 3: Transforming the OR action combinator (part II)

Now a term rewriting system is generated:

$< or(A_1, A_2); C, [Z, [T, B, S]] > \quad \Rightarrow \quad < A_1; conv_1; fact_{or}(A_2); C, [[[B, T]|Z], [T, B, S]] >$

$< fact_{or}(A); C, [Z, [[T, B], [completed, T_1, B_1, S_1]]] > \quad \Rightarrow \quad < C, [Z, [completed, T_1, B_1, S_1]] >$

$< fact_{or}(A); C, [Z, [[T, B], [failed, [], [], S]]] > \quad \Rightarrow \quad < A; C, [Z, [T, B, S]] >$

$< conv_1; C, [[B, T]|Z], [O, T_1, B_1, S_1]] > \quad \Rightarrow \quad < C, [Z, [[T, B], [O, T_1, B_1, S_1]]] >$

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

$or(A_1, A_2) \quad \Rightarrow \quad \overline{or}(A_1, A_2); A_1; conv_1; fact_{or}(A_2)$

$fact_{or}(A) \quad \Rightarrow \quad \overline{fact}_{or}(A)$

$conv_1 \quad \Rightarrow \quad \overline{conv}_1$

And the following abstract machine rules:

$< \overline{or}(A_1, A_2); C, [Z, [T, B, S]] > \quad \Rightarrow \quad < C, [[[B, T]|Z], [T, B, S]] >$

$< \overline{fact}_{or}(A); C, [Z, [[T, B], [completed, T_1, B_1, S_1]]] > \quad \Rightarrow \quad < C, [Z, [completed, T_1, B_1, S_1]] >$

$< \overline{fact}_{or}(A); C, [Z, [[T, B], [failed, [], [], S]]] > \quad \Rightarrow \quad < A; C, [Z, [T, B, S]] >$

$< \overline{conv}_1; C, [[B, T]|Z], [O, T_1, B_1, S_1]] > \quad \Rightarrow \quad < C, [Z, [[T, B], [O, T_1, B_1, S_1]]] >$

$$\begin{aligned}
x_1 = \{x : \quad & k \notin \mathcal{L}(x), \exists n \in \mathcal{L}(x) \cup \mathcal{R}(x) : n < k \\
& and\ \exists m : m \in \mathcal{L}(x), m > k \\
& or\ m \in \mathcal{R}(x), m \geq k\} \\
x_2 = \{x : \quad & k \notin \mathcal{R}(x), \exists n : n \in \mathcal{L}(x), n \leq k \\
& or\ n \in \mathcal{R}(x) : n < k \\
& and\ \exists m : m \in \mathcal{L}(x), m > k \\
& or\ m \in \mathcal{R}(x), m > k\} \\
x_3 = \{x : \quad & k \in \mathcal{C}(x), k \notin \mathcal{L}(x) \\
& and\ \exists n \in \mathcal{L}(x) \cup \mathcal{R}(x) : n < k\}
\end{aligned}$$

Finally, if v is empty, then we do not change the stack. Note, that allocating temporary variables before factorization would destroy common initial segments. Consider the two 2BIG rules defining the OR action combinator in Figure 2. The left hand sides of the first precondition of both rules are equal and would be part of the common segment. In the first rule no variable is temporary, in the second rule the variables T and B are temporary. Allocating these variables in the second rule would change the left side of its first precondition and it would no longer be part of the common segment of the rules.

3.5 Sequentialization

Next we will transform the rules, such that the environment on the right side of a precondition is equal to the environment on the left side of the subsequent precondition. Furthermore the environment on the right side of the last precondition is equal to the environment on the right side of the conclusion. More precisely, a rule of the form

$$\frac{c_1, e_1 \rightarrow e_1' \quad \ldots \quad c_n, e_n \rightarrow e_n'}{c, e \rightarrow e'}$$

is transformed into

$$\frac{\begin{array}{l} c_1, e_1 \rightarrow e_1' \quad p_1, e_1' \rightarrow e_2 \\ \ldots \\ p_{n-1}, e_{n-1} \rightarrow e_n \quad c_n, e_n \rightarrow e_n' \quad p, e_n' \rightarrow e' \end{array}}{c, e \rightarrow e'}$$

and we add the rules

$$\begin{gathered}
p_1, e_1' \rightarrow e_2 \\
\vdots \\
p_{n-1}, e_{n-1}' \rightarrow e_n \\
p, e_n' \rightarrow e
\end{gathered}$$

Figure 2: Transforming the OR action combinator (part I)

In the 2BIG specification the following rules define the action combinator *or*. In the rule the environment 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*.

$$\frac{A_1,[T,B,S]\rightarrow[completed,T_1,B_1,S_1]}{or(A_1,A_2),[T,B,S]\rightarrow[completed,T_1,B_1,S_1]}$$

$$\frac{A_1,[T,B,S]\rightarrow[failed,[],[],S_1] \qquad A_2,[T,B,S_1]\rightarrow[O_2,T_2,B_2,S_2]}{or(A_1,A_2),[T,B,S]\rightarrow[O_2,T_2,B_2,S_2]}$$

There are no side conditions, so the above rules are next factorized:

$$\frac{A_1,[T,B,S]\rightarrow[O_1,T_1,B_1,S_1] \qquad fact_{or}(A_2),[[B,T],[O_1,T_1,B_1,S_1]]\rightarrow E}{or(A_1,A_2),[T,B,S]\rightarrow E}$$

$$fact_{or}(A),[[B,T],[completed,T_1,B_1,S_1]]\rightarrow[completed,T_1,B_1,S_1]$$

$$\frac{A,[T,B,S]\rightarrow[O,T_1,B_1,S_1]}{fact_{or}(A),[[B,T],[failed,[],[],S]\rightarrow[O,T_1,B_1,S_1]}$$

Now the stack (Z) is introduced and temporary variables are allocated, e.g. in the first rule T,B are allocated on the stack, because they do not occur on the right side of the first precondition.

$$\frac{A_1,[[[T,B]|Z],[T,B,S]]\rightarrow[[[T,B]|Z],[O_1,T_1,B_1,S_1]] \qquad fact_{or}(A_2),[Z,[[B,T],[O_1,T_1,B_1,S_1]]]\rightarrow[Z,E]}{or(A_1,A_2),[Z,[T,B,S]]\rightarrow[Z,E]}$$

$$fact_{or}(A),[Z,[[B,T],[completed,T_1,B_1,S_1]]]\rightarrow[Z,[completed,T_1,B_1,S_1]]$$

$$\frac{A,[Z,[T,B,S]]\rightarrow[Z,[O,T_1,B_1,S_1]]}{fact_{or}(A),[Z,[[B,T],[failed,[],[],S]]\rightarrow[Z,[O,T_1,B_1,S_1]]}$$

Next these rules are sequentialized:

$$\frac{\begin{array}{l}A_1,[[[T,B]|Z],[T,B,S]]\rightarrow[[[T,B]|Z],[O_1,T_1,B_1,S_1]]\\ conv_1,[[[T,B]|Z],[O_1,T_1,B_1,S_1]]\rightarrow[Z,[[B,T],[O_1,T_1,B_1,S_1]]] \quad fact_{or}(A_2),[Z,[[B,T],[O_1,T_1,B_1,S_1]]]\rightarrow[Z,E]\end{array}}{or(A_1,A_2),[Z,[T,B,S]]\rightarrow[Z,E]}$$

$$fact_{or}(A),[Z,[[B,T],[completed,T_1,B_1,S_1]]]\rightarrow[Z,[completed,T_1,B_1,S_1]]$$

$$\frac{A,[Z,[T,B,S]]\rightarrow[Z,[O,T_1,B_1,S_1]]}{fact_{or}(A),[Z,[[B,T],[failed,[],[],S]]\rightarrow[Z,[O,T_1,B_1,S_1]]}$$

$$conv_1,[[[T,B]|Z],[O_1,T_1,B_1,S_1]]\rightarrow[Z,[[B,T],[O_1,T_1,B_1,S_1]]]$$

$$\mathcal{L}(x)=[i : x\ occurs\ in\ e_i]$$
$$\mathcal{R}(x)=[i : x\ occurs\ in\ e'_i]$$
$$\mathcal{C}(x)=[i : x\ occurs\ in\ c_i]$$

Now we convert the preconditions in the rule as follows: Let $c,[s_1,e_1]\rightarrow[s_2,e_2]$ be the k-th precondition in the rule. Then it is converted into $c,[[v|s_1],e_1]\rightarrow[[v|s_2],e_2]$ where $v=(x_1\cup x_2\cup x_3)-\mathcal{V}(c_0)$ and

Now $\mathcal{C}$ is replaced by:

$$\frac{\text{seg} \quad c_{1j},e_{1j} \rightarrow \overline{e} \quad \kappa,\overline{e} \rightarrow e'}{c,e \rightarrow e'}\theta$$

where e' is a new variable name

$$\frac{c_{1(j+1)},e_{1(j+1)} \rightarrow e'_{1(j+1)} \quad \cdots \quad c_{1m_1},e_{1m_1} \rightarrow e'_{1m_1}}{\kappa,e_{1j} \rightarrow e'_1}\theta$$
$$\vdots$$
$$\frac{c_{n(j+1)},e_{n(j+1)} \rightarrow e'_{n(j+1)} \quad \cdots \quad c_{nm_1},e_{nm_1} \rightarrow e'_{nm_1}}{\kappa,e_{nj} \rightarrow e'_n}\theta$$

3.3 Stack Introduction

In the next step the environments in the rules are extended by a stack. This stack will be used later to store temporary variables[1]. A rule of the form

$$\frac{c_1,e_1 \rightarrow e'_1 \quad \ldots \quad c_n,e_n \rightarrow e'_n}{c,e \rightarrow e'}$$

is converted into

$$\frac{c_1,[s,e_1] \rightarrow [s,e'_1] \quad \ldots \quad c_n,[s,e_n] \rightarrow [s,e'_n]}{c,[s,e] \rightarrow [s,e']}$$

where s is a new variable name.

3.4 Allocation of Temporary Variables

Intuitively a variable is called temporary in a rule, if there is an intermediate environment where the variable does not occur. The rules are transformed, such that variables are passed in the environment from the precondition of their first occurrence to the precondition of their last occurrence. More precisely:

Definition: A variable X is *temporary* in a rule,

1. if it does not occur in the commands of the left-hand side of the conclusion and
2. its leftmost occurrence is in the right side of a precondition or the left side of the conclusion and
3. it occurs at least a second time in another precondition or in the right-hand side of the conclusion and
4. one of the following conditions is true
   - there is a precondition different from the one in 2. where it occurs in the right hand side but not in the left hand side
   - there are two successive preconditions (the second being different from the one in 2.) where the variable does not occur in the right side of the first precondition, but in the left side of the second
   - it occurs in the right side of the conclusion, but not in the right side of the last precondition

Consider the rule:

$$\frac{c_1,e_1 \rightarrow e'_1 \quad \ldots \quad c_n,e_n \rightarrow e'_n}{c_0,e_0 \rightarrow e'_{n+1}}$$

We define for every variable name x in the rule, the lists of its right and left hand side occurrences in environments and commands.

[1]An optimization not discussed here stores the results of function calls on the stack.

Side conditions of the form *not* $p(t_1, \ldots, t_n)$ are converted into

$$a(x_1, \ldots, x_k), [y_1, \ldots, y_m] \rightarrow false$$

where a is a new instruction symbol. Finally we generate a new rule

$$a(x_1, \ldots, x_k), [y_1, \ldots, y_m] \rightarrow p'(t1, \ldots, t_n)$$

where p' is the characteristic function of the predicate p, $\{x_1, \ldots, x_k\} = \mathcal{V}(c) \cup \mathcal{V}(t_1, \ldots, t_n)$ and $\{y_1, \ldots, y_m\} = \mathcal{V}(t_1, \ldots, t_n) - \mathcal{V}(c)$. Dividing the variables occurring in the side condition this way, guarantees, that only variables occurring in the commands of the conclusion are arguments of the new instruction. The remaining variables are passed in the environment.

3.2 Factorization

A transformation, which converts determinate inductive rules into deterministic rules has been proven correct by daSilva [1]. The following factorization transformation can be regarded as an instance of this transformation. We also extended the transformation to sets of more than two conflicting rules. Basically the transformation computes for a set of conflicting rules, a new rule, which has the common initial preconditions of the conflicting rules and a precondition, which calls a new instruction, as its preconditions. For each conflicting rule, we add a rule defining the new instruction, which has the rest of the preconditions of the conflicting rule as its preconditions.

By $=_\alpha$ we will denote equality of terms and formulae modulo renaming of variables. Two rules are conflicting, if they have the same left hand sides in their conclusions. Let $\mathcal{C}$ be the largest set of conflicting rules with respect to the same left hand side:

$$\frac{\mathrm{c}_{11}, \mathrm{e}_{11} \rightarrow \mathrm{e}'_{11} \quad \cdots \quad \mathrm{c}_{1\mathrm{m}_1}, \mathrm{e}_{1\mathrm{m}_1} \rightarrow \mathrm{e}'_{1\mathrm{m}_1}}{\mathrm{c}_1, \mathrm{e}_1 \rightarrow \mathrm{e}'_1}$$
$$\vdots$$
$$\frac{\mathrm{c}_{\mathrm{n}1}, \mathrm{e}_{\mathrm{n}1} \rightarrow \mathrm{e}'_{\mathrm{n}1} \quad \cdots \quad \mathrm{c}_{\mathrm{nm}_1}, \mathrm{e}_{\mathrm{nm}_1} \rightarrow \mathrm{e}'_{\mathrm{nm}_1}}{\mathrm{c}_\mathrm{n}, \mathrm{e}_\mathrm{n} \rightarrow \mathrm{e}'_\mathrm{n}}$$

where $c_1, e_1 =_\alpha \ldots =_\alpha c_n, e_n$.
Let θ be a renaming of variables and j be the largest integer, such that for all $p, q \in \{1, \ldots, n\}$: $c_{pj}\theta = c_{qj}\theta$ and for all $k < j$: $(c_{pk}, e_{pk} \rightarrow e'_{pk})\theta = (c_{qk}, e_{qk} \rightarrow e'_{qk})\theta$.
Let us call the ordered set (wrt k) of the latter transitions the common segment *seg*. Furthermore let $\overline{e}$ be the common term of $e'_{1j}, \ldots, e'_{nj}$, i.e. $\overline{e} = e'_{1j} \odot \ldots \odot e'_{nj}$:

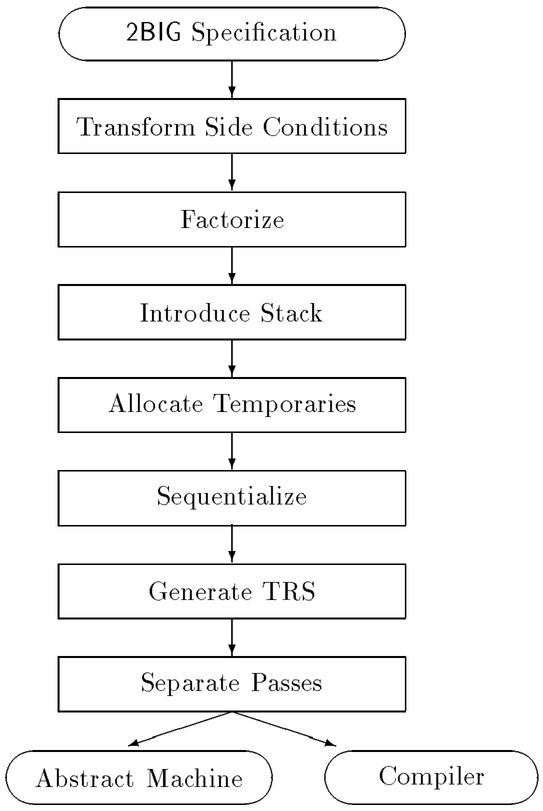
$$e_1 \odot e_2 = \begin{cases} e_1\theta & \text{if } e_1\theta = e_2\theta \text{ are the same variable name} \\ f(d_1, \ldots, d_n) & \text{if } e_1\theta = f(a_1, \ldots, a_n),\ e_2\theta = f(b_1, \ldots, b_n) \text{ and } d_i = a_i \odot b_i \\ c(d_1, \ldots, d_n) & \text{if } e_1\theta = c(a_1, \ldots, a_n),\ e_2\theta = c(b_1, \ldots, b_n) \text{ and } d_i = a_i \odot b_i \\ x & \text{otherwise} \end{cases}$$

where x is a new variable name

Let ι be a new instruction symbol and

$V_1 = \bigcup_{k=1}^{n} \mathcal{V}(c_{k(j+1)}, \ldots, c_{km_k}, e_{k(j+1)}, \ldots, e_{km_k}, e'_k)\theta$
$V_2 = \mathcal{V}(seg, c, e)\theta \cup \bigcup_{k=1}^{n} \mathcal{V}(c_{kj}, e_{kj})\theta$
$V = (V_1 \cap V_2) - \mathcal{V}(\overline{e})\theta$
$\kappa = \iota(x_1, \ldots, x_m)$ where $x_i \in V$

Figure 1: Overview of the Transformations



while-loops into a compiler. We formally defined similar transformations and implemented them in Prolog. Applying our system to the above mentioned toy language yields similar results. The goal of our work is to apply the method to specifications of realistic programming languages and thus detecting missing links, insufficiencies and possible optimizations of the transformations. We decided to use action notation as such a programming language, because it offers a rich set of primitives underlying both imperative and functional programming languages.

2 Two-Level Big-Step Semantics (2BIG)

First we give the syntax of 2BIG, a language designed to write natural semantics specifications. The language combines the structural approach of natural semantics [8] with the idea to split general and implementation details by the use of a separately given interpretation for function symbols [10, 14, 16].

$$\boxed{\begin{array}{lcll}
T & ::= & f(T^*) \mid c(T^*) \mid x & \textit{x variable symbol} \\
\tilde{T} & ::= & c(\tilde{T}^*) & \textit{c constructor symbol} \\
C & ::= & T & \textit{f function name} \\
\tilde{C} & ::= & \tilde{T} & \textit{p predicate name} \\
E & ::= & T & \\
\tilde{E} & ::= & \tilde{T} & \\
S & ::= & C, E \rightarrow E & \\
\tilde{S} & ::= & \tilde{C}, \tilde{E} \rightarrow E & \\
Q & ::= & p(T^*) \mid \mathit{not}\ p(T^*) & \\
J & ::= & S \mid Q & \\
R & ::= & \frac{\mathrm{J}^*}{\tilde{\mathrm{S}}} &
\end{array}}$$

Judgements are transitions of the form $C, E_1 \rightarrow E_2$ or side conditions of the form $p(T^*)$ or $\mathit{not}\ p(T^*)$. We will use the term *left hand side* to refer to C, E_1 in a transition and the term *right hand side* to refer to E_2. In a rule of the form $\frac{J^*}{\tilde{S}}$ the judgements above the line are called *preconditions* and the judgement below the line is called the *conclusion*. We will denote the variables in a term t by $\mathcal{V}(t)$. Furthermore we adopt the notation for list constructors from Prolog. Specifications in 2BIG have to be determinate [1], i.e. whenever two rules have conclusions, which unify with a goal at most one of the rules can be successfully applied. The restriction to determinate rule sets is important, because determinate rule sets can be converted into deterministic ones, i.e. at most one rule will have a conclusion, which unifies with a goal. Deterministic rules can be converted into term rewriting rules and finally these rewrite rules can be pass separated into rewrite rules for a compiler and an abstract machine. In the next section the transformations are discussed in more detail.

3 Generating Compilers and Abstract Machines from 2BIG Specifications

An overview of the system is given in Figure 1. Since the system transforms specifications by successively applying transformations, we will present the transformations in the order of their application. Actually the transformations have been devised in reversed order. Starting from the pass separation transformation we tried to remove restrictions on the input specifications by transforming a more general class of specifications into the class of input specifications. This process finally lead to determinate 2BIG specifications. Note, that after each transformation we have an executable specification again.

3.1 Transformation of Side Conditions

This transformation converts side conditions into judgements. Let $c, e \rightarrow e'$ be the conclusion of a rule. Side conditions of the form $p(t_1, \ldots, t_n)$ are converted into

$$a(x_1, \ldots, x_k), [y_1, \ldots, y_m] \rightarrow \mathit{true}$$

Automatic Generation of a Compiler and Abstract Machine for Action Notation

(Preliminary Results)

Stephan Diehl
FB 14 - Informatik, Universität des Saarlandes,
Postfach 15 11 50, 66041 Saarbrücken,
GERMANY , diehl@cs.uni-sb.de

Abstract

We present a system, that generates a compiler and abstract machine from a Natural Semantics specification of a programming language. First an overview of the system and the transformations involved are given. Then we apply the system to a specification of Actress Action Notation. As an example we trace the transformations of rules for an action combinator. The resulting compiler and abstract machine can be used as a basis for a compiler generator based on Action Semantics. Finally we discuss future work.

Contents

1 Introduction **2**

2 Two-Level Big-Step Semantics (2BIG) **3**

3 Generating Compilers and Abstract Machines from 2BIG Specifications **3**
3.1 Transformation of Side Conditions 3
3.2 Factorization 5
3.3 Stack Introduction 6
3.4 Allocation of Temporary Variables 6
3.5 Sequentialization 8
3.6 Generation of Term Rewriting Systems 9
3.7 Pass Separation 10

4 Transforming a 2BIG specification of Action Notation **10**

5 Future Work **12**

6 Conclusions **13**

1 Introduction

Abstract machines provide an intermediate target language for compilation. First the compiler generates code for the abstract machine, then this code can be further compiled into real machine code or it can be interpreted. By dividing compilation into two stages abstract machines increase portability and maintainability of compilers. Usually abstract machines are designed in an ad-hoc manner often based on experience with other abstract machines. But also some systematic approaches have been investigated. One of those is based on partial evaluation of example programs [9, 15, 3]. Another approach is to use pass separation transformations [7]. John Hannan [6] introduced a pass separation transformation, which splits a set of term rewriting rules representing an abstract interpreter into two sets of term rewriting rules: the first set represents a compiler into an abstract machine language, the second set represents an abstract machine. Since rewrite rules are a poor language to specify interpreters, Stephen McKeever [11] extended Hannan's transformations to determinate inductive rules. In McKeever's framework the factorization algorithm of Fabio daSilva [1] plays a central role. By hand McKeever transformed a natural semantics specification for an imperative toy language with

Automatic Generation of a Compiler and an Abstract Machine for Action Notation

Stephan Diehl

Technischer Bericht A 03/95

FB 14 - Informatik
Universität des Saarlandes, Postfach 15 11 50
66041 Saarbrücken , GERMANY
Phone: ++49-681-3023915
diehl@cs.uni-sb.de