OASIS: An Optimizing Action-based Compiler Generator

Peter Ørbæk*

Computer Science Department, Aarhus University
Ny Munkegade Bldg. 540, DK-8000 Aarhus C, Denmark

Abstract. Action Semantics is a new and interesting foundation for semantics based compiler generation. In this paper we present several analyses of actions, and apply them in a compiler generator capable of generating efficient, optimizing compilers for procedural and functional languages with higher order recursive functions. The automatically generated compilers produce code that is comparable with code produced by handwritten compilers.

1 Introduction

Semantics based compiler generation has long been a goal in computer science. Automatic generation of compilers from semantic descriptions of programming languages relieves programmers and language theorists from much of the burden of writing compilers.

We describe the OASIS (Optimizing Action-based Semantic Implementation System) compiler generator, and especially the analyses that provide the information enabling the code generator to produce good quality code.

The generated compilers expand a given abstract syntax tree to the equivalent action by way of the action semantics for the language. All analyses are applied to the expanded action. The system is capable of generating compilers for procedural, functional (lazy and eager) and object oriented languages. After analysis, the action is translated to native SPARC code. For further details, see [18].

A short introduction to Action Notation is given first, and in the following section we describe a type-checker for actions, whose *raison d'être* is to allow us to dispense with all run-time type checks.

We then proceed to describe the various analyses that are carried out on the type checked action. Most of the analyses are set up in an abstract interpretation framework. The analyses annotate the action with approximate information about its run-time behavior.

The results of the analyses are used by a code generator, generating code for the SPARC processor. The code generator also employs a couple of optimization techniques on its own, namely a storage cache used to avoid dummy stores and reloads, and a peephole optimizer responsible for filling delayslots and removing no-op code.

Finally we compare the performance of the generated compilers for a procedural and a functional language with handwritten compilers for similar languages, and relate our results to previous approaches to compiler generation.

* The authors Internet address: `poe@daimi.aau.dk`

The results are very encouraging as our automatically generated compilers emit code that performs within a factor 2 of code produced by handwritten compilers. This is a major performance enhancement in relation to earlier approaches to compiler generation based on Action Semantics [20, 19, 3], as well as compared to other semantics based compiler generators.

2 Action Notation

Action Semantics is a formalism for the description of the dynamic semantics of programming languages, developed by Mosses and Watt [17]. Based on an order-sorted algebraic framework, an action semantic description of a programming language specifies a translation from abstract terms of the source language to Action Notation.

Action Notation is designed to allow comprehensible and accessible semantic descriptions of programming languages; readability and modularity are emphasized over conciseness. Action semantic descriptions scale up well, and considerable reuse of descriptions is possible among related languages. An informal introduction to Action Notation, as well as the formal semantics of the notation, can be found in [17].

The semantics of Action Notation is itself defined by a structural operational semantics, and actions reflect the gradual, stepwise, execution of programs. The performance of an action can terminate in one of three ways: It may *complete*, indicating normal termination; it may *fail*, to indicate the abortion of the current alternative; or it may *escape*, corresponding to exceptional termination which may trapped. Finally, the performance of an action may *diverge*, ie. end up in an infinite loop.

Actions may be classified according to which *facet* of Action Notation they belong. There are five facets:

- the *basic* facet, dealing with control flow regardless of data.
- the *functional* facet, processing *transient* information, actions are *given* and *give* data.
- the *declarative* facet, dealing with bindings (*scoped information*), actions *receive* and *produce* bindings.
- the *imperative* facet, dealing with loads and stores in memory (*stable information*), actions may *reserve* and *unreserve* cells of the storage, and change the contents of the cells.
- the *communicative* facet, processing *permanent information*, actions may *send* and *receive* messages communicated between processes.

In general, imperative and communicative actions actions are *committing*, which prevents backtracking to alternative actions on failure. There are also hybrid actions that deal with more than one facet. Below are some example action constructs:

- 'complete': the simplest action. Unconditionally completes, gives no data and produces no bindings. Not committing.
- 'A_1 and A_2': a basic action construct. Each sub-action is given the same data as the combined action, and each receives the same bindings as the combined construct. The data given by the two sub-actions is tupled to form the data given

by the combined action, (the construct is said to be *functionally conducting*). The performance of the two sub-actions may be interleaved.

- 'A_1 or A_2': a basic action construct, represents non-deterministic choice between the two sub-actions. Either A_1 or A_2 is performed. If A_1 fails without committing A_2 is performed, and vice versa.
- 'store Y_1 in Y_2': an imperative action. Evaluates the yielder Y_1 and stores the result in the cell yielded by Y_2. Commits and completes when Y_1 evaluates to a storable and Y_2 evaluates to a cell.

An action term consists of constructs from two syntactic categories, there are action constructs like those described above, and there are yielders that we will describe below. Yielders may be evaluated in one step to yield a value. Below are a few example yielders:

- 'sum(Y_1, Y_2)': evaluates the yielders Y_1 and Y_2 and forms the sum of the two numbers.
- 'the given D#n': picks out the n'th element of the tuple of data given to the containing action. Yields the empty sort nothing unless the n'th item of the given data is of sort D.
- 'the D stored in Y': provided that Y yields a cell, it yields the intersection of the contents of that cell and the sort D.

As an example we give below an action semantics for a simple call-by-value λ-calculus with constants.

2.1 Abstract Syntax

needs: **Numbers/Integers**(integer), **Strings**(string) .
grammar:

```
(1)  Expr = [[ "lambda" Var "." Expr ]] | [[ Expr "(" Expr ")" ]] |
            [[ Expr "+" Expr ]] | integer | Var .
(2)  Var = string .
```

2.2 Semantic Functions

includes: **Abstract Syntax** .
introduces: evaluate _ .

```
 •   evaluate _ :: Expr → action .
(1)  evaluate I:integer = give I .
(2)  evaluate V:Var = give the datum bound to V .
(3)  evaluate [[ "lambda" V:Var "." E:Expr ]] =
         give the closure abstraction of
         |furthermore bind V to the given datum#1
         |hence evaluate E .
(4)  evaluate [[ E1:Expr "(" E2:Expr ")" ]] =
         |evaluate E1 and evaluate E2
         then enact application the given abstraction#1 to the given datum#2 .
```

(5) evaluate ⟦ E_1:Expr "+" E_2:Expr ⟧ =
  | evaluate E_1 and evaluate E_2
  then give the sum of them .

2.3 Semantic Entities

includes: Action Notation .

- datum = abstraction | integer | □ .
- bindable = datum .
- token = string .

3 Type Checking

The main purpose of the type-checker for action notation is to eliminate the need for run-time type-checking. If we hope to gain a run-time performance comparable to traditional compiled languages such as C and Pascal, we need to eliminate run-time type-checks, otherwise values would have to carry tags around identifying their type, and we would immediately suffer the penalty of having to load and store the tags as well as the actual values. We are thus lead to choose a wholly static type system.

Our type-checker is related to the one given by Palsberg in [20], but our type-checker is also capable of handing unfoldings that are not tail-recursive. This imposes some problems, since fixpoints have to be computed in the type lattice. Like Palsberg's type-checker, our type-checker can be viewed as an abstract interpretation of the action over the type lattice.

The type-checker has been proved *safe* with respect to the structural operational semantics of a small subset of Action Notation [18], but we will not go into details about the proof here, we just give the structure of the proof. It should be straightforward, but tedious, to extend the proof to the larger subset accepted by OASIS.

First a type inference system for a small subset of Action Notation is defined. It is shown that the inference system has the Subject Reduction property with respect to the operational semantics of Action Notation. Second, the type checking *algorithm* is proved *sound* with respect to the inference system. Finally subject reduction and soundness are combined to prove the safety of the type checker. Below we state the main safety result:

> *The type-checker is safe in the sense that, if the type-checker infers a certain type for the outcome of an action then, when that action is actually performed, the outcome indeed has the inferred type.*

As a corollary to the safety property, we have proved that all run-time type-checks can be omitted.

4 Analyses

The main reason for the good run-times that we are able to achieve for the produced code, is the analyses that we apply to the action generated from the semantics. The analyses consist of the following stages:

- **forward analysis**, incorporating constant analysis, constant propagation, commitment analysis and termination analysis.
- **backwards flow analysis**, used to shorten the lifetime of registers.
- **heap analysis**, determines which frames can be allocated on the stack, and which need to be allocated on the heap.
- **tail-recursion detection**, checking whether unfoldings are tail-recursive or not.

4.1 Forward Flow Analysis

The forward flow analysis is essentially an abstract interpretation of the action over a product of several complete lattices. The various parts of the analysis are interleaved in order to obtain better results than would be possible had the analyses been done one after the other.

The forward analyses can be divided up into the following parts:

- **constant analysis**, determines whether bound values are static or dynamic.
- **constant propagation and folding**, propagates constants and folds expressions with constant arguments into constants.
- **commitment analysis**, approximates the commitment nature of the action.
- **termination analysis**, approximates the termination mode of the action.

All of the analyses are set up in an abstract interpretation framework [5]. They are all essentially intra-procedural, so each abstraction is not analyzed in relation of all of its enactions (calls).

Constant propagation and folding is a well-known technique often used in compilers to reduce constant expressions to constants. There is nothing special about our constant propagation technique for actions. For example if the two arguments propagated to a "sum" yielder are constant, the sum is folded into a constant at compile time, and propagated further as a constant. The other parts of the forward analysis are more interesting.

4.1.1 Constant Analysis Since the constant analysis is integrated with the constant propagation, we use the following lattice of abstract values for this part of the analysis:

$$\mathsf{SD} = (\{(\mathsf{Static},\ v),\ \mathsf{Dynamic}\},\ \leq)$$

where for all values v : $(\mathsf{Static},\ v) \leq \mathsf{Dynamic}$.

The above lattice differs from the traditional lattice used in binding time analyses (eg. in [1]) by incorporating the statically known value with the Static tag. This only buys us a marginal benefit, but it is simply the obvious thing to do when the constant analysis and constant propagation are integrated.

A binding of a constant (Static) value need not have space allocated in the frame of the enclosing abstraction, as the bound value can be inserted statically wherever it is referenced. Bindings of dynamic values are associated with a cell (a memory location) in the relevant frame, and when a value is bound at run-time, it is stored in the cell, and it is retrieved from that cell whenever the bound value is referenced.

Arguments to abstractions are assumed to be dynamic. We do not attempt to do inter-procedural constant analysis. The need for such an analysis is not too great for ordinary imperative languages, where fewer and larger procedural abstractions dominate. Also, an intra-procedural analysis can be done more efficiently. We avoid the compile-time performance problems often associated with inter-procedural abstract interpretations.

Likewise, the analysis assumes that the contents of memory cells are dynamic. All loads and stores in Action Notation go through pointers, and storing a value to wherever a dynamic pointer points may over-write the contents of any cell of the same type. This problem with dynamic pointers is usually known as *aliasing* problems in traditional compilers. Since the performance of sub-actions may be interleaved, it is hard to guarantee that the contents of a cell has not been over-written by an unknown value at any given point of the performance.

4.1.2 Commitment Analysis Actions may or may not *commit*. If an action commits it means that it has made some irreversible change to the state of the machine, such as having stored a value in a cell or having sent a message to another process.

We are interested in knowing whether an action may commit and subsequently fail (ie. err) within an "A_1 or A_2" construct. If this is the case, the "or" can't trap the failure, a run-time error should be indicated and the program stopped (in the absence of commitment, an alternative action may be performed). In the CANTOR system [20, 19], a significant amount of run-time is used to check for such committed failures. Our analysis is able to statically determine the possibility of committed failures in most cases, thus much fewer run-time checks need to be inserted. The lattice used by the commitment analysis looks like this:

$$\begin{tikzcd}
 & \texttt{may-err} \arrow[d, dash] & \\
 & \texttt{may-commit} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{never-commits} \arrow[dr, dash] & & \texttt{always-commits} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Commitment of ground actions such as "store Y_1 in Y_2" is determined by the type checker. Commitment of combined actions such as "A_1 and A_2" is determined by commitment of the sub-actions and the results of the termination analysis on the sub-actions. For unfoldings a fixed point is computed.

4.1.3 Termination Analysis The termination analysis computes approximate knowledge about the termination mode of sub-actions. There are many benefits to be drawn from such knowledge. For example, suppose we have the action "A_1 then A_2". If the termination analysis is able to guarantee that the sub-action A_1 always fails, no code need to be generated for A_2!

The termination analysis abstractly interprets the action over the power-set of the four possible termination modes (*complete, fail, escape* and *diverge*) ordered by subset inclusion:

$$\mathsf{CFED} = (\mathcal{P}\{\mathsf{AC}, \mathsf{AF}, \mathsf{AE}, \mathsf{AD}\}, \subseteq)$$

The termination mode of ground actions is determined by the type checker and influenced by the propagated constants. The termination mode of combined actions is determined by the termination mode of the sub-actions.

The termination- and commitment analyses have been formally specified in [18], although soundness still remains to be proven.

4.2 Backwards Analysis

The backwards analysis is used to shorten the lifetime of transient values. This analysis traces the data-flow backwards and increases counters in the abstract compile-time representations of values each time such values are used, ie. stored to memory, written to standard output or passed as parameters to an abstraction.

During code generation, the same counters are decreased at each point of usage and when the counter reaches zero, the register holding the value can safely be discarded for eventual re-use. This analysis is similar to the computation of *live variables* in traditional optimizing compilers.

4.3 Heap Analysis

Since abstractions are first class values in action notation, they can be given as transient data, returned from abstractions and stored in memory. As we deal with statically scoped languages, we need to provide abstractions with their correct static environment when they are enacted (called).

In traditional languages with first class abstractions, such as Scheme, all frames (or activation records) are typically allocated on the heap and it is up to a garbage collector to release the associated memory when it is no longer used. In order to avoid spending lots of time doing garbage collection, and to avoid heap allocated frames for programs in traditional imperative languages such as Pascal, we employ the heap analysis[2]. The heap analysis is yet another abstract interpretation of the action, this time over the following domain:

$$(\mathcal{P}(\{\mathsf{SA}, \mathsf{PC}\}), \subseteq)$$

The analysis traverses the action and marks each abstraction with an element from the above domain as explained in the following:

SA stands for *stores abstraction,* it means that the abstraction may store, give, or escape with an abstraction, ie. an abstraction may leak out of scope. PC stands for *provides closure,* it means that the abstraction provides (part of) the closure for another abstraction, ie. it has a syntactically nested abstraction. Only if an abstraction is marked $\{\mathsf{SA}, \mathsf{PC}\}$ need the corresponding frame be allocated on the heap. Note that the top-most

[2] This analysis was initially called *closure analysis* for obvious reasons, but that term has a more specific and different meaning in Copenhagen, so it was renamed.

or global frame can always be allocated on the stack as it will exist until the program terminates.

Thanks to this analysis, an action semantics for full Pascal will never give rise to code needing heap allocated closures, as it is impossible for a procedure in Pascal to leak out of scope.

4.4 Tail Recursion

In order to implement standard `while` loops efficiently by the "unfolding" construct, we need to be able to detect tail-recursive unfoldings, so as not to incur the overhead of a procedure call for each iteration of the loop. The action semantic equation for a `while` construct would typically look something like:

```
execute ⟦ "while" E:Expression "do" S:Statements ⟧ =
    evaluate E then
    |unfolding
    ||check the given truth-value then execute S then unfold
    ||or
    ||check not the given truth-value .
```

In full generality, "unfold" may cause a recursive call. The tail-recursion detector traverses the body of the unfolding and marks "unfolds" as tail-recursive or recursive depending on whether any part of the loop-body may be executed after the "unfold". If there is just one recursive "unfold" in the body of a loop, then all "unfolds" in that loop are treated as recursive.

5 Code Generation

The code generator generates assembly code for the SPARC processor from the action tree annotated by the preceding analyses. The assembly code is generated in one pass, and registers are allocated on an as-needed basis.

As much of the code generator as possible is kept machine-independent, to facilitate easy porting of the code generator to other RISC processors. One machine-independent part of the code generator is the storage cache. It serves the purpose of minimizing the number of load and store operations in basic blocks. When a value is loaded from a known memory location into a register, an association between the register and the location is kept, such that a later load from the same address can be coded as a cheap register copy. Storing the contents of a register in a known memory location keeps the association between the location and the register in the same way, and the actual store is delayed until the last possible moment within the same basic block to avoid two stores to the same location just after each other. (This may not be entirely beneficial on a RISC architecture, where loads and stores should be spread out, but it was easier to implement than a full graph-coloring register allocation algorithm).

A machine-dependent part of the code generator is the peephole optimizer. A peephole optimizer is a traditional optimization technique, that is often used to remove dummy instruction sequences and to simplify instructions. Our peephole optimizer does not attempt to eliminate all dummy instructions, but is geared towards fixing deficiencies in the code generated by our specific code generator.

The code generator is pretty intricate, as there are lots of special cases to consider when one tries to generate good code for a realistic machine such as the SPARC. Perhaps a code generator generator such as `iBurg` [8] could be used clean this up.

6 Overview

The action compiler and compiler generator consists of many parts written in different languages[3]. This section gives an overview of the different parts and their interaction.

The compiler generator (`gencomp`) takes an action semantics written as a Scheme [4] program, and produces a compiler written in Perl [24]. Scheme was chosen because it was easy to implement a few macros in Scheme that make it painless to write a semantics using Scheme syntax. Also, it was felt that not too much time should be spent on this part of the compiler generator, as work is in progress that will make it possible to write action semantics in the ASF+SDF system [11].

The generated compiler driver (or front-end) is written in Perl for ease of implementation. The driver parses command-line options, calls the different parts of the compiler and takes care of cleaning up if something goes wrong, such as a syntax error in the given program etc.

The compiler takes a textual representation of an abstract syntax tree (AST) for a program in the source language, and produces, if all goes well, executable code for the SPARC processor. The AST could easily be produced by, say, a YACC or Bison generated parser for the source language.

The first step of the compiler is to massage the input AST into something resembling a Scheme program, and combine it with the Scheme representation of the semantics for the source language.

The second step runs a Scheme interpreter on the semantics and the munged program, and writes a textual representation of the action corresponding to the program to a file. Currently, the free `scm` implementation of the Scheme standard [4] is used.

Step three runs the action compiler on the produced action, and produces assembly code for the SPARC processor. This is the major step of the process. The action compiler consists of approximately 10,000 lines of C++ code [21], plus a lexical analyzer generated by Flex and an action parser generated by Bison.

The fourth step of the compiler assembles the output from the action compiler and links the object module with a small run-time support library providing primitive input/output routines. The run-time library is written in traditional C [10].

The produced object program reads from standard input and writes to standard output.

[3] The OASIS system is available by anonymous ftp from `ftp.daimi.aau.dk` in the directory `/pub/action/systems/`

7 Comparisons

Here we compare the performance of our compiler generator with handwritten compilers and other approaches to compiler generation. We consider two example languages: HypoPL and FunImp.

The procedural language HypoPL contains integers, booleans, arrays, the usual control structures (while-loops and conditionals) and generally nested procedures. The syntax and semantics for HypoPL can be found in [12, 19, 18]. The functional (eager) language FunImp contains higher order recursive functions as well as mutable data. The syntax and semantics for FunImp is derived from the language considered in [22], and is defined in [18].

All our timings, except for the run-times of the generated code, are made on an ordinary 33 MHz 386-based PC with 20 MB RAM, running the Linux operating system (version 0.99). The generated SPARC code was run on a Sun Microsystems SparcStation ELC running SunOS 4.1.1.

Generating a compiler for Lee's HypoPL language [12, 13] takes 0.8 seconds. Using the generated compiler to compile the HypoPL bubblesort program takes 3.9 seconds. As explained in a previous section, this involves running the Perl interpreter, the Scheme interpreter and the action compiler, and the result is an assembly file suitable for the SPARC assembler. The assembly code consists of roughly 250 instructions, ie. 1000 bytes when assembled.

Comparing these figures with what Palsberg obtained with the CANTOR system [20, 19] shows that the compilers we generate are two orders of magnitude faster than his, and that the code size is also two orders of magnitude smaller than his. It should be noted that Palsberg's tests were also run on a Sun SparcStation ELC.

The tables below show some results from using the generated HypoPL compiler to compile some example programs (the same programs as used in [20, 19]):

- **bubble:** A bubblesort program, bubblesorts 500 integers.
- **sieve:** The sieve of Eratosthenes, finds all primes below 512, repeated 400 times.
- **euclid:** Euclid's method of finding the greatest common divisor of two numbers (1023 and 37), repeated 30,000 times.
- **fib:** Computes the 46'th Fibonacci number 10,000 times. (The 46'th number in the series is the largest that will fit in a 32 bit twos complement integer.)

The table below lists the compile times of various programs. The first column lists the time it takes to compile the HypoPL program to assembly, the second column lists the time it takes to compile the action generated from the HypoPL program, and the last column lists the time it takes to compile an equivalent C program with optimization turned on. All times are in seconds.

Program	HypoPL	Action	C-opt
bubble	3.9	0.9	0.6
sieve	3.4	0.9	0.5
euclid	2.3	0.4	0.4
fib	2.1	0.4	0.3

The figures above indicate that something could be gained by integrating the processing of the semantic functions with the action compiler, instead of relying on a Scheme interpreter to expand the program to an action.

The generated HypoPL compiler is on average 6.5 times slower than the hand-written C compiler. Much of this slowdown stems from the Scheme interpreter. The action compiler itself compiles an action within a factor two of the time it takes to compile the equivalent C program. This is what one would expect, as the action is at least twice as large (textually) as the corresponding C program.

The "code size" column in the table below is a simple line count of the generated assembly files. The actual number of instructions is smaller because of a little overhead, such as assembler directives, labels and so forth. All times are in seconds. The fourth column gives the run-time for an equivalent program written in C and compiled with the GNU C compiler (`gcc 2.4.3`). The last column states the run-time for the C program compiled with full optimization turned on.

Program	Code size	Run-time	C-runtime	C-opt
bubble	254	0.4	0.4	0.2
sieve	211	1.2	0.7	0.3
euclid	144	2.1	1.4	0.7
fib	93	0.8	0.7	0.5

The above table shows that the run-times for all four programs are within a factor 1.7 of code generated by a hand-written C compiler. If we let the C compiler do its best at optimizing the program, our code is still at most 4 times slower. The main reason why our code is so much slower than optimized C code is the lack of a global register allocator. Another reason is that HypoPL allows general nesting of procedures, something that C doesn't. This has an impact on performance, since a HypoPL compiler (in the absence of a corresponding analysis) cannot take the same shortcuts as a C compiler can when accessing variables.

Unrolling the `sieve` program a number of times, to obtain a source program ten times as large (539 lines) yield compile times (19 seconds) about ten times longer than for the small program (2 seconds), as one would expect, since the analyses are of linear complexity. Keeping the actual amount of computation constant, we get the same run-times for the small and large program. The code size scales linearly too, of course.

The figures show that code generated with the OASIS system is about two orders of magnitude faster than code generated with the CANTOR system

7.1 FUNIMP versus `scm`

Here we make a performance comparison between Scheme and the generated FUNIMP compiler. The example program is a recursive Fibonacci function.

The first column below shows the number of seconds it takes to compute the result in interpreted Scheme, the second column is for the Scheme program compiled to C and then to machine code by the Hobbit [23] compiler. The last column shows the run-time for the OASIS-compiled FUNIMP program. Again our results are within a factor two of a hand-written compiler.

scm	Compiled Scheme	FUNIMP
84.8	3.4	5.7

The main difference between the code we generate for `fib` and the code that Hobbit/C generates, is that our implementation of the conditional is less than optimal, due to the symmetric nature of the "or" action construct and our non-optimal implementation of the "check" construct. Our code actually computes the truth value, whereas C need only test the condition. Comparing against an optimized, equivalent hand-written C program, shows that our code is about 3 times slower.

7.2 Lee and Pleban

Comparing performance against Lee's system [12, 13] is difficult since it ran on much slower hardware than what is available today. Comparing the time that it took for that system to compile a HypoPL program to the time it takes for OASIS on more modern hardware would be unfair. Also, traditional compiler technology has improved since his comparisons with the traditional compilers of then, making a comparison based on those relative figures difficult.

Lee's system is based on High Level Semantics, where the static semantics is separated from the dynamic semantics, and he explicitly gives a so-called micro semantics tailored for the processor. Giving a new micro-semantics in his system would equal writing a new code generator part to the action compiler. If one were to write a micro-semantics targeting the SPARC processor, then it would be realistic to assume that code produced by a HypoPL compiler generated from the high level semantics system could be as good as the code produced by a HypoPL compiler generated by the OASIS system.

However, it seems that all optimizations in Lee's system happens at the micro-semantic level, hence a new micro semantics will be difficult to write.

7.3 Kelsey and Hudak

In [9] Kelsey and Hudak describe their compiler generator based on denotational semantics. In their system one writes denotational descriptions of languages in a variant of Scheme, and the system then performs several transformations on the resulting Scheme program, eventually arriving at assembly code for the Motorola 68020 processor.

They evaluate the performance of their system by comparing code produced by a generated Pascal compiler with code produced by the standard Pascal compiler on the Apollo workstation they used. The quality of the produced code is as good as what the standard Pascal compiler can generate, all performance figures lie within a factor 1.5 of the Pascal-generated code. Assuming that the standard Pascal compiler on the Apollo is comparable to a standard C compiler, one will have to say that the performance of their system is on a par with the OASIS system.

Apart from being based on denotational semantics, the main difference between their system and OASIS, is that their system *transforms* the meta-language (Scheme) until they reach something that is close enough to assembly to warrant a mechanical substitution from Scheme terms to assembly code. In OASIS the meta-language (Action

Notation) is not transformed, but merely annotated by the various phases of analysis, and then ultimately an intricate code generator is invoked to generate assembly.

7.4 Bondorf and Palsberg

Using the same subset of Action Notation as in [20], Bondorf and Palsberg in [2] present another compiler generator based on Action Semantics. The compiler generator partially evaluates a Scheme representation of the action generated from the semantics. The generated compilers are compared with compilers generated by the CANTOR system. In comparison with the CANTOR system, run-times of the produced Scheme code are improved by at most a factor of 4, including a hypothetical factor 5 that the authors think they would achieve, had they used a Scheme compiler instead of an interpreter. Since the produced object code from the OASIS system is two orders of magnitude faster than what the CANTOR generated compilers produce, our system is clearly superior to this partial evaluation approach to compiler generation.

7.5 Actress

Comparing performance against the ACTRESS system is difficult since only one small test program with timings is given in [3]. For the system that they had implemented at the time, they write that the code they produce (C code) is 69 times slower than the equivalent Pascal program compiled with a standard compiler. This is certainly slower than our system. With certain mechanical optimizations, that were not implemented at the time of their article, they improve performance to within a factor two of the Pascal compiler. No timings are given for how long it takes to compile a program with the generated compilers.

The Actress approach to action compilation is closer to the approach by Kelsey and Hudak than it is to ours, in that they *transform* the action generated by the semantics to gain better run-times.

8 Concluding Remarks and Future Work

We have described several analyses based on Action Semantics, and have shown how they can be applied in a compiler generator capable of generating compilers that produce code comparable to code produced by handwritten compilers for similar languages.

Even though Action Semantics was developed from a semantic perspective without regard for compilability and run-time efficiency, we have demonstrated that efficient compilers can be automatically generated from Action Semantic descriptions.

However, there are various shortcomings of the current version of the system. The type system is probably too strict, and some sort of type inference like the system by Even and Schmidt [7] would be an advantage, if it could be modified in a way that would allow us to dispense with – most or all – run-time type checks. Moreover, many useful data-types are not easily expressible in the system, such as lists and records. Again the type system would have to be extended to cater for them.

There is still ample room for improvements of the code quality. The contents of memory cells should be tracked, and loop optimizations such as strength reduction could be applied. One possible way to obtain better code would be to transform the action tree to some other internal form better suited for low level optimizations, such as RTL (Register Transfer Language) [16, 15] or structured RTL [14].

A few experiments have been made with the specification and generation of compilers for object oriented languages. A small language with classes, objects, block structure and inheritance has been specified and a compiler has been generated. We simply employ the ability of OASIS to handle higher order abstractions to model objects and methods. However, the current system is not capable of resolving non-virtual method-calls at compile time, as further analysis would be needed to accomplish that.

Work is currently going on to formally specify the various analyses described in this paper, and to prove their safety with respect to the operational semantics of Action Notation.

Further work could go in the direction of using the results of analyses on actions to say something about the source program. It would also be useful to analyze the semantic equations themselves (akin to the work by Doh and Schmidt [6]), this could perhaps cut down on the time it takes to compile actions to assembly. Generally it would be advantageous to analyze as much as possible in the compiler *generation* phase, as opposed the *compilation* phase. Typically one will apply the generated compilers more often that the compiler generator.

9 Acknowledgements

I want to thank Peter D. Mosses for encouragements and guidance in the preparation of the article. I also want to thank Ole L. Madsen for reading a draft of the paper. I must also thank the anonymous referees for useful and guiding feedback.

References

1. A. Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP 90)*, volume 432 of LNCS, pages 70–87, Copenhagen, May 1990. Springer-Verlag.
2. A. Bondorf and J. Palsberg. Compiling Actions by Partial Evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, 1993.
3. D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an Action Semantics Directed Compiler Generator. In *Proceedings of the 1992 Workshop on Compiler Construction, Paderborn, Germany*, volume 641 of LNCS. Springer-Verlag, 1992.
4. W. Clinger and J. R. (editors). Revised^4 Report on the Algorithmic Language Scheme. Technical report, MIT, 1991.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

6. K.-G. Doh and D. A. Schmidt. Extraction of Strong Typing Laws from Action Semantics Definitions. In B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP 92)*, volume 582 of LNCS, pages 151–166, Rennes, February 1992. Springer-Verlag.
7. S. Even and D. A. Schmidt. Type Inference for Action Semantics. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP 90)*, volume 432 of LNCS, pages 118–133, Copenhagen, May 1990. Springer-Verlag.
8. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. Technical report, AT&T Bell Labs, 1992.
9. R. Kelsey and P. Hudak. Realistic Compilation by Program Transformation. In *Proceedings of the 16'th ACM Symposium on Principles of Programming Languages*, pages 281–292, January 1989.
10. B. W. Kernighan and D. M. Richie. *The C Programming Language*. Prentice-Hall, 1978.
11. P. Klint. A meta-environment for generating programming environments. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490 of LNCS, pages 105–124, Mierlo, September 1989. Springer-Verlag.
12. P. Lee and U. F. Pleban. A Realistic Compiler Generator on High-Level Semantics. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.
13. P. Lee and U. F. Pleban. An Automatically Generated, Realistic Compiler for an Imperative Programming Language. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 222–232, June 1988.
14. C. McConnel. *Tree-Based Code Optimization*. PhD thesis, University of Illinois Urbana Champaign, March 1992. Draft.
15. C. McConnel and R. E. Johnson. Using SSA Form in a Code Optimizer. Technical report, UIUC, 1991.
16. C. McConnel, J. D. Roberts, and C. B. Schoening. The RTL System. Technical report, UIUC, October 1990.
17. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992. Number 26 in the Cambridge Tracts in Theoretical Computer Science series.
18. P. Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Computer Science Department, Aarhus University, Denmark, September 1993.
19. J. Palsberg. A Provably Correct Compiler Generator. In B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP 92)*, volume 582 of LNCS, pages 418–434, Rennes, February 1992. Springer-Verlag.
20. J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Computer Science Department at Aarhus University, January 1992.
21. B. Stroustrup and M. A. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
22. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
23. T. Tammet. Hobbit: A Scheme to C compiler. Unpublished, (available by ftp from `nexus.yorku.ca:/pub/scheme`), 1993.
24. L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1991.