ACTRESS: an Action Semantics Directed Compiler Generator

Deryck F. Brown*, Hermano Moura**, David A. Watt

Department of Computing Science,
University of Glasgow, Glasgow, G12 8QQ, UK.

Abstract. We report progress on the development of ACTRESS, a compiler generator based on action semantics. It consists of a number of modules, written in SML, that can be composed to construct either an action notation compiler or a simple compiler generator. We also outline current and future developments that will improve the quality of the generated compilers.

1 Introduction

We define a *compiler generator* to be a tool that constructs a compiler automatically, given a syntactic and semantic description of the source language. This definition emphasizes the fact that the compiler is not written by a programmer, but generated from a formal description of the language. Ideally, this formal description is one that was written by the language designer and is consulted by users of the language.

We can generate an efficient syntactic analyser automatically from a syntactic description, using tools such as LEX and YACC. However, generating a code generator from a semantic description is much more difficult, and so far this problem has eluded satisfactory solution.

The key efforts have been attempts to generate compilers from denotational semantic descriptions [3, 7, 11]. The generated compiler translates the source program to a λ-expression, then reduces the latter as much as it can; the residual λ-expression is the "object program". At run-time the residual λ-expression is applied to the program's input, and when reduced to normal form gives the program's output.

There are many fundamental problems with this approach. Reduction of the λ-expression is slow (both at compile-time and at run-time). Environments and stores are passed around as arguments, just like ordinary values. Moreover, simple environment and store operations are represented by λ-functions, so the number of λ-reductions is enormous. Thus the generated compilers are hugely inefficient, typically exhibiting a performance penalty of about 1000 relative to a hand-crafted compiler (both at compile-time and at run-time).

The only hope of generating realistic compilers from denotational descriptions is for the compiler generator to "understand" the special properties of environments, stores, and continuations. For example, Schmidt has formulated sufficient conditions

* Supported by SERC, UK.
** Supported by CNPq, Brazil. On leave from Caixa Econômica Federal, Brazil.
Authors' e-mail address: {deryck, moura, daw}@dcs.glasgow.ac.uk.

for single-threading of the store arguments; under these conditions all store aguments can be mapped to a global store variable [9]. Similarly, he has attempted to analyse the use of environment arguments, in order to discover whether the scopes of bindings and the lifetimes of variables permit stack allocation [8]. But the analyses are complicated, and it is unlikely that they can be extended beyond toy languages.

Action semantics [4, 5, 12] seems to be a more promising basis for compiler generation. An action-semantic description in effect specifies a translation from the source language to *action notation*. Although designed primarily for readability and modularity, action notation has a number of properties that make it suitable for the purposes of compiler generation. The action primitives and combinators correspond quite closely to the operational concepts in terms of which languages are implemented. The store is by definition single-threaded, and bindings are by definition scoped. These properties of action notation eliminate the need for Schmidt's storage analysis, and simplify the analysis required to determine, for example, whether stack allocation is possible. Finally, action notation has numerous algebraic laws that provide a rigorous foundation for code-improving transformations.

We have constructed a preliminary version of an action semantics directed compiler generator, ACTRESS, and we are currently developing a more sophisticated version. This paper is a progress report on our work. Section 2 briefly summarizes action notation (and may be skipped by readers already familiar with the notation). Section 3 gives an overview of the compiler generator and its component modules: the action notation parser, sort checker, code generator, and interpreter, and the "actioneer" generator. Section 4 takes a closer look at the sort checker, which infers useful information about the flow of data among actions. Section 5 describes the code generator, which translates action notation to C. Section 6 gives some performance measurements on our generated compilers, and describes current and future developments of our work. Section 7 concludes by comparing our work with contemporary compiler generation projects.

2 Action Notation

An *action* is a computational entity, which can be *performed*. When performed, an action might *complete* (i.e., terminate normally), or *fail* (i.e., terminate abnormally), or *diverge*, (i.e., not terminate at all). An action may use data supplied to it by other actions; and it can supply data to other actions if it completes.

Actions have several *facets*, which differ in the nature and lifetime of the data involved. In the *functional facet*, an action may use and/or give *transients*. Transients are structured as tuples of data, with each datum being individually labeled. Transients disappear unless used immediately. In the *declarative facet*, an action may use and/or produce *bindings*. Bindings are structured as sets of identifier–datum associations. Bindings generally propagate throughout a designated action, which is their scope. In the *imperative facet*, an action may use and/or change *storage*. Storage is structured as a set of cells, each containing a single datum. Stored data are stable, in that they remain available for use by any action until overwritten.

Action notation provides a number of *action primitives*, *action combinators*, and *data operations*. An action primitive represents a single step in the computation, and

Table 1. Some action primitives

Primitive	Informal meaning	Example
complete	Completes immediately (i.e., does nothing).	
fail	Fails immediately.	
give *d*	Gives a single datum *d*.	give sum (the integer, 2)
give *d* label #*n*	Gives a single datum *d*, labeled *n*.	give the integer label #2
bind *k* to *d*	Produces a single binding, of identifier *k* to datum *d*.	bind "n" to the integer
store *d* in *c*	Stores datum *d* in cell *c*.	store 0 in the cell bound to "x"
allocate a *S*	Finds an unreserved cell of sort *S*, reserves it, and gives it.	allocate an [integer] cell
enact *a*	Performs the action incorporated by the abstraction *a*.	enact the abstraction bound to "p"

generally has an effect in (at most) one facet. An action combinator combines one or two sub-actions into a composite action, and determines the flow of control and flow of data between the sub-actions. An action may use data operations to access the data supplied to it. The most common action primitives, action combinators, and data operations are summarized in Tables 1, 2, and 3, respectively.

An action semantic description of a small imperative language, MINI-△ [12], is given in Appendix A. It is structured like a denotational semantic description, with semantic functions and semantic equations, but the denotations are expressed in action notation.

Space permits only a very brief and incomplete explanation of action notation here. For a comprehensive account of action notation, together with a formal specification, see Mosses [4]; for a gentler introduction, see Watt [12].[3]

3 Overview of ACTRESS

ACTRESS supports a well-defined subset of action notation that is rich enough to write semantic descriptions of interesting programming languages. The subset includes all the notation listed in Tables 1–3.

ACTRESS consists of a number of modules, which may be composed in various ways. These modules are briefly explained below. In the following sections two of the modules are discussed in more detail.

The *action notation parser* parses a textual action, and translates it to the corresponding action notation abstract syntax tree (*action tree*). It is an LALR(1) parser, and was generated using ML-YACC.

[3] For historical reasons, the action notation used in ACTRESS differs slightly from that described in the cited texts.

Table 2. Some action combinators

Combinator	Informal meaning	Example
A_1 or A_2	Performs either A_1 or A_2. If the chosen sub-action fails, the other sub-action is chosen.	give the value or give the value stored in the cell
A_1 else A_2	Tests a given truth value, and then performs A_1 if it is true or A_2 if it is false.	give 1 else give 0
A_1 and A_2	Performs both A_1 and A_2 collaterally. Any transients given by A_1 and A_2 are merged. Any bindings produced by A_1 and A_2 are merged.	bind "x" to the cell and store 0 in the cell
A_1 and then A_2	Performs A_1 and A_2 sequentially. Otherwise behaves like 'A_1 and A_2'.	give the value stored in the cell and then store 0 in the cell
A_1 then A_2	Performs A_1 and A_2 sequentially. Transients given by A_1 are given to A_2.	allocate an [integer] cell then store 0 in the cell
A_1 hence A_2	Performs A_1 and A_2 sequentially. Bindings produced by A_1 propagate to A_2.	bind "h" to 3600 hence give product (24, the integer bound to "h")
A_1 before A_2	Performs A_1 and A_2 sequentially. Bindings produced by A_1 and A_2 are accumulated.	bind "n" to 7 before bind "m" to successor (the integer bound to "n")
furthermore A	Performs A. Bindings produced by A override the received bindings.	furthermore bind "x" to the argument
unfolding A	Performs A iteratively. Dummy action 'unfold', whenever encountered inside A, is replaced by A.	unfolding give successor (the integer) then unfold

The *action notation sort checker* traverses the action tree, inferring the sorts of the transients and bindings that flow into and out of each action, and checking whether the action will fail. The sort checker decorates the action tree with sort information.

The *action notation code generator* translates the decorated action tree to a C object program. Sort information is used to guide register allocation, to generate code for any run-time sort checks, and especially to guide the translation of recursive actions and abstractions. The object program passes transients in registers, explicitly manipulates bindings, and operates on a global store.

These three modules when composed form the first of our tools, the *action notation compiler*. This compiles an action to a C object program. The object program may then be compiled and executed in the usual way.

The *actioneer generator* accepts the dynamic semantic description of a source language $\mathcal{L}$, and generates an $\mathcal{L}$ "actioneer". The latter is a module that translates

Table 3. Some data operations

Operation	Informal meaning	Example
the S	The given transient datum. It must be of sort S.	the truth-value
the $S\#n$	The given transient datum labeled n. It must be of sort S.	the integer#2
the S bound to k	The datum currently bound to identifier k. It must be of sort S.	the cell bound to "x"
the S stored in c	The datum currently contained in cell c. It must be of sort S.	the integer stored in the cell
abstraction A	The abstraction that incorporates action A.	abstraction |give successor(the integer)
closure a	The abstraction a with the current bindings supplied to the incorporated action.	closure abstraction |store 0 in the cell bound to "x"
a with d	The abstraction a with the transient datum d supplied to the incorporated action.	the abstraction bound to "f" with the argument

the abstract syntax tree (AST) of an $\mathcal{L}$ program to its denotation, an action tree.

Composing an $\mathcal{L}$ parser with a generated $\mathcal{L}$ actioneer, the action notation sort checker, and the action notation code generator gives us an $\mathcal{L}$ compiler. The structure of this generated compiler is shown in Figure 1.

The *action notation interpreter* interprets an action tree and displays the action's outcome. The outcome includes the transients, bindings and storage produced by the action, if it completes, or an indication of failure otherwise. The interpreter closely follows the structure of Mosses' operational semantics of action notation [4]. Moreover, it implements nearly all of action notation. A full description can be found in Moura [6].

The action notation interpreter can be composed with the action notation parser to allow the user to perform actions directly — a valuable tool for students of action notation. Alternatively, the interpreter can be composed with an $\mathcal{L}$ parser and an $\mathcal{L}$ actioneer to generate an $\mathcal{L}$ interpreter — a useful prototyping tool.

4 The Action Notation Sort Checker

The action notation sort checker is similar in principle to an ordinary type checker, but is in fact significantly more sophisticated. Its purpose is to traverse the action tree and decorate each action with the tagged sorts of that action's input and output data. This is done for both the functional facet (transients) and the declarative facet (bindings). The imperative facet (storage) is not currently analysed.

The sort information is represented by record types similar to those introduced by Wand, and applied by Even and Schmidt to their own simplified version of action notation [10]. Each action is decorated by four record schemes, representing input

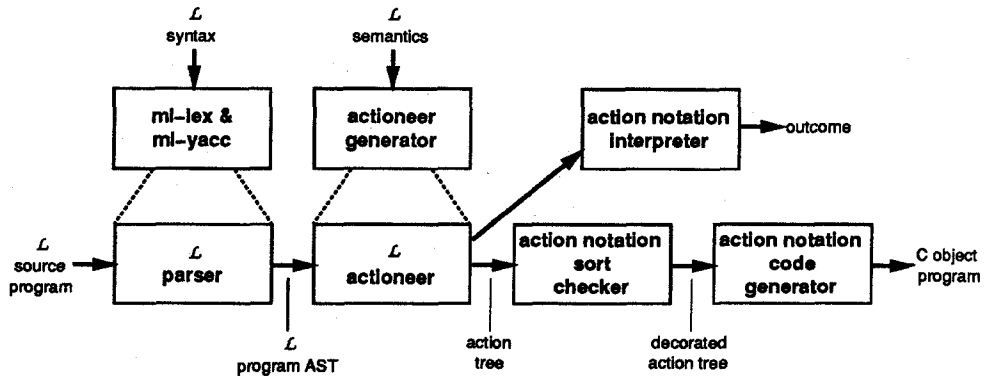

Fig. 1. The structure of a compiler generated by ACTRESS

and output transients, and input and output bindings. For example, a record for transients (data tagged by labels) might be:

{0 : integer, 1 : truth-value}

and a record for bindings (data tagged by identifiers):

{"n" : 7, "m" : integer, "z" : [integer] cell}

In action notation, individual values (such as 7) are treated as singleton sorts, and the sort checker can sometimes infer such values. In the example above, n is bound to a known integer of value 7, but m is bound to some unknown integer, and z to some unknown cell.

Using sort information, it is possible to detect certain failing actions before code generation. An action that is guaranteed to fail when performed is rewritten by the sort checker to a single "fail" action. In turn, this can be used to eliminate occurrences of the "or" combinator, since "fail" is the unit for "or". For example, in MINI-△ (Appendix A), the action denoting the expression x is:

give the value bound to "x" or
give the value stored in the cell bound to "x"

Here one of the sub-actions must fail, since x cannot be bound to both a value and a cell. Once the sort checker has inferred the sort of the datum bound to x, it replaces the "or" action by the appropriate sub-action.

The sorts inferred by the sort checker have been specified using a set of inference rules, somewhat analogous to the type inference rules for a programming language. Our sort inference algorithm is based on the one given in Even and Schmidt [10]. However, it has been enhanced in several ways. It represents both transients and bindings using record schemes, which leads to a more regular structure in the sort checker. Additionally, it handles a larger subset of action notation, including recursive actions, non-deterministic choice, and abstractions (which are essential for writing useful language descriptions).

The sort checker consists of three passes. The first pass decorates the action tree with record schemes; the second pass removes all the sort variables present in the record schemes, and reduces the sorts to a canonical form; the third pass simplifies the action tree where possible, and marks the places where run-time sort checks are required. For more details see Brown [1].

5 The Action Notation Code Generator

The actual translation of actions into C object code is done by the code generator. An action is translated to a C statement (-sequence); a term yielding a datum is translated to a C expression. In the generated code, transients and bindings are passed in registers. A register allocation discipline is necessary: the flow of data between actions must guide the allocation and deallocation of registers. The code generator is also guided by information received from the sort checker.

Figure 2 shows some of the translation rules built into the code generator. For example, the action "complete" is translated to the C null statement.

Each transient datum is contained in a special kind of register called a *d-register*. For example, the action "give *d* label #*n*" is translated to an assignment of the value of *d* to a d-register allocated at translation-time ($\mathtt{d}_i$). The translation process must note the association between *n* and $\mathtt{d}_i$. Thus, "the *S*#*n*" is translated to a fetch from the d-register associated with the label *n*. A run-time sort check may be necessary here to guarantee that the content of the register is of sort *S*; the code generator is warned by the sort checker and generates the necessary code.

At run-time, a second kind of register called a *b-register* points to a set of bindings. The translation of "bind *k* to *d*" is just an assignment of a single binding to a b-register, allocated at translation-time. Such a binding is built by calling an auxiliary run-time function, `BIND(`*k*`,`$[\![d]\!]$`)`. The translation of "the *S* bound to *k*" is just a call to another auxiliary function, $\mathtt{BOUND}(k,\mathtt{b}_j)$, that looks up what datum is bound to token *k* in register $\mathtt{b}_i$ (which is determined at translation-time).

Storage is represented by an array, `storage`, which explains the translation rules for "store *d* in *c*" and "the *S* stored in *c*".

"A_1 and then A_2" is translated to sequential code: the code for A_1 followed by the code for A_2. The generated code for A_1 must not reuse any d-register still to be read by the translation of A_2. Similarly, the generated code for A_2 must not reuse any d-register written by A_1. The bindings produced by the subactions are merged. Sort information is used to achieve an efficient translation. For example, if A_1 produces non-empty bindings in register $\mathtt{b}_i$, and A_2 produces non-empty bindings in $\mathtt{b}_j$, then the translation-time function *merge_bindings()* generates something like "$\mathtt{b}_k$ = `MERGE(`$\mathtt{b}_i$`,`$\mathtt{b}_j$`);`". But if either A_1 or A_2 produces no bindings, *merge_bindings* generates no code at all.

"unfolding *A*" is translated to a loop, in which each tail-recursive occurrence of the dummy action "unfold" becomes a jump back to the beginning of the loop. This gives us an efficient translation of source-language iterators. (ACTRESS does not at present handle non-tail-recursive occurrences of "unfold".)

An abstraction is represented at run-time by a C structure with three fields: a pointer to a C function (the translation of the incorporated action), together with a datum and a set of bindings (the ones supplied to the incorporated action).

(1) ⟦ complete ⟧ ⇝ ;

(2) ⟦ give d label $\#n$ ⟧ ⇝ d_i = ⟦d⟧ ; (associating n with d_i)

(3) ⟦ the $S\#n$ ⟧ ⇝ d_i (where n is associated with d_i)

(4) ⟦ bind k to d ⟧ ⇝ `b`$_j$ = `BIND (`k`,` ⟦d⟧`);`

(5) ⟦ the S bound to k ⟧ ⇝ `BOUND (`k`,` `b`$_j$`);`

(6) ⟦ store d in c ⟧ ⇝ `storage[`⟦c⟧`]` = ⟦d⟧;

(7) ⟦ the S stored in c ⟧ ⇝ `storage[`⟦c⟧`]`

(8) ⟦ A_1 and then A_2 ⟧ ⇝ ⟦A_1⟧
⟦A_2⟧
merge_bindings()

(9) ⟦ unfolding A ⟧ ⇝ `l`$_i$`:`
⟦ A ⟧

(10) ⟦ unfold ⟧ ⇝ `goto l`$_i$`;` (provided that unfold is tail-recursive)

Fig. 2. Some translation rules

Appendix B shows an example MINI-△ program (Figure 3), its corresponding program action after sort checking (Figure 4), and its corresponding object program (Figure 5). By comparing these, the reader should be able to see how the translation is performed.

The run-time environment defines the data representation as a tagged union of relevant data types (`bool`, `int`, etc). The tag allows run-time sort checks to be performed where necessary. The run-time environment also contains functions corresponding to data operations (such as `SUM` and `PRODUCT`), sort-checking functions, auxiliary functions (such as `BIND` and `BOUND`), and storage management functions (`ALLOCATE_A_CELL` and `DEALLOCATE_THE_CELL`).

6 Current and Future Developments

The preliminary version of ACTRESS has been used to generate compilers for:

- the small functional language NANO-ML, which has integers, higher-order functions, let-expressions, and conditional expressions;
- the small imperative language MINI-△, which has integers, truth values, assignment, if- and while-commands, procedures and parameters (see Appendix A).

Together, these languages are representative of the range of language concepts that we currently handle.

The compilation of an example MINI-△ program is shown in Appendix B (Figures 3–5). We have measured the running time of this program, and compared it with the running time of a similar PASCAL program. The results are summarized in Table 4. All figures have been expressed relative to the running time of the PASCAL program, which was compiled using the SUN compiler `pc`. Thus the figures in Table 4 may be interpreted as the performance penalty of a generated MINI-△ compiler relative to a hand-crafted compiler.

The performance penalty of 69 is very much better than the classical compiler generators, but still not satisfactory. We are currently working on transformations

Table 4. Performance figures for the generated MINI-△ compiler

Object program	Running time (relative)
MINI-△ object program	69
... after action transformations	27
... after action transformations and object-code improvements	2
PASCAL object program	1

that will dramatically reduce the performance penalty. We have performed these transformations manually (but mechanically) on our example program. As shown in Table 4, these transformations reduce the performance penalty to 2. Taking into account the fact that the generated compiler's target language is C, rather than machine code, this is very gratifying. The rest of this section summarizes the work in progress.

Action notation was designed to be suitable for describing the semantics of a wide range of programming languages: imperative and functional, statically and dynamically typed, statically and dynamically scoped. In order to be as general as possible, storage allocation, sort checking, and bindings are all dynamic. This is so even for a source language that happens to be statically typed and scoped. Our current work is directed to discovering and exploiting important properties of the source language from its semantic description, such as whether it is statically typed and/or statically scoped.

A hand-crafted compiler for a statically-scoped language, such as MINI-△, would eliminate all bindings. Consider the MINI-△ declaration "`const n ~ 7`": here the identifier `n` is bound to a statically known integer value (7), so the compiler simply replaces each applied occurrence of `n` by that value. Now consider the declaration "`const m ~ i+j`": here the identifier `m` is bound to a statically unknown integer value, so the compiler generates code to compute the unknown value and store it at a known address, and replaces each applied occurrence of `m` by a fetch from that address. This method works uniformly for identifiers bound to known and unknown values, variables, procedures, etc. Thus, whether identifiers are bound to known or unknown data, their bindings can be eliminated from the object program.

The next version of the action notation code generator will apply this binding-elimination transformation. In effect it will eliminate the declarative facet from the program action. The action notation sort checker already supplies the necessary information about which identifiers are bound to known and unknown data. For example, if the sort checker infers the bindings {"n" : 7, "m" : integer}, we can tell that n is bound to a known integer but m is bound to an unknown integer. In the program action of Figure 4, the action "give 1000000 then bind "n" to the value" can be eliminated, and the term "the value bound to "n"" replaced by "1000000".

Binding elimination is possible only if the source language is statically scoped. We are formulating a test on the source language's semantics that will determine whether the language is statically scoped or not. A simple sufficient condition is that the "closure" operation is applied to every abstraction as soon as it is formed — closure abstraction (...) — since the closure operation "freezes" the bindings used

by the incorporated action. The MINI-Δ semantics of Appendix A clearly exhibits static scoping.

Another problem that we are currently studying is storage allocation. The "allocate" action performs dynamic storage allocation, i.e., by default all variables are heap variables. We want the generated compiler to use static or stack allocation wherever possible, because these are inherently faster than heap allocation. Moreover, static and stack allocation assign a known relative address to each variable — unlike heap allocation — so our binding-elimination transformation will work better.

We can classify the "allocate" actions by analyzing the flow of control. Action notation makes this analysis rather easy. If an "allocate" action is performed exactly once (i.e., neither selectively nor iteratively) in the program action, it is allocating a global variable that can be statically allocated. If an "allocate" action is performed exactly once inside an abstraction, it is allocating a local variable. All local variables in a particular abstraction can be allocated together in a frame, and each of them will have a known relative address. Any "allocate" action not so classified must be presumed to be allocating a heap variable.

In the program action of Figure 4, the action "allocate an [integer] cell" is allocating a global variable. Thus it can be replaced by something like "give global cell 0". The composite action "give global cell 0 then bind "x" to the cell" can now be eliminated, and each occurrence of "the cell bound to "x"" replaced by "global cell 0".

Allocating local variables together in frames is a necessary but not sufficient condition for stack allocation. If abstractions are first-class values in the source language, the frames themselves must be allocated in the heap. Thus we will have to formulate a test on the source language's semantics to determine whether abstractions are first-class or not. The MINI-Δ semantics of Appendix A does in fact permit stack allocation.

Binding elimination, combined with static and stack allocation, and another transformation (transient elimination) not discussed here, will allow us to reduce the performance penalty from 69 to 27 in the case of our example program.

The above may be viewed as transformations on the program action. Inspection of the object code reveals opportunities for improvement at that level too. At present the object code contains many calls to trivial run-time functions, such as `MAKE_INTEGER`, `SUM`, and `PRODUCT`, whose only purpose is to construct and operate on *tagged* data. It will be straightforward to expand these calls into in-line code. Furthermore, the code generator generates a run-time sort check only where there is a risk of the sort check failing. If *no* run-time sort checks are generated, the code generator should generate an object program in which data are untagged. The combined effect of these simple improvements will be to reduce the performance penalty from 27 to 2 in the case of our example program, as shown in Table 4.

Figure 6 shows the object code we obtained by manually (but mechanically) performing all these transformations. It is quite similar to the object code of a hand-crafted compiler.

As well as the efficiency of the generated compiler's object code, we are addressing the efficiency of the generated compiler itself. If the source language is known to be statically typed, there is no need to test at compile-time whether run-time sort checks will be needed.

More importantly, at present every generated compiler incorporates the action notation sort checker. The latter is very general but cumbersome, making three passes over the (large) action tree, and employing unification. For a language with a simple type system, such as MINI-Δ, a much simpler and more efficient sort checker can be constructed.

We are examining afresh how we should process the given action semantics of the source language $\mathcal{L}$. The existing actioneer generator accepts the given semantics, warts and all. A re-designed generator will perform consistency checks on the given semantics, and generate a sort checker specific to $\mathcal{L}$, as well as generating an $\mathcal{L}$ actioneer. The generated $\mathcal{L}$ sort checker will decorate the source program's AST, employing a simple sort-checking algorithm if $\mathcal{L}$ is found to have a simple type system. The generated $\mathcal{L}$ actioneer will then expand the decorated AST to a decorated action tree. We estimate that these improvements would speed up the sort-checking phase of the generated MINI-Δ compiler by a factor of about 10.

7 Conclusion

Our work on action semantics directed compiler generation should be seen in the context of the programming language life cycle [12]. Language development proceeds in several stages: design, specification, prototyping, and compiler construction (not necessarily in that order). We see action semantics as a means of integrating all stages of language development. Its excellent pragmatic properties (being easy to write, to modify, and to read) make action semantics highly suitable for documenting a new and evolving language design. Its operational flavour and clean structural properties (facets) make it suitable for generating prototypes and compilers.

Current work on semantics directed compiler generation seems to be concentrating on applying partial evaluation to denotational semantics. It is not clear how successful this line of work will prove to be, given that the effectiveness of partial evaluation in general depends critically on the style of the program being partially evaluated. In any case, denotational semantics has poor pragmatic qualities as a language development tool: denotational semantic descriptions of real programming languages are notoriously hard to write, to modify, and to read.

Also worth mentioning here is Lee's "high-level semantics" [2]. This is a method of constructing compilers using semantic algebras, which are not unlike action notation. Using this method Lee has constructed compilers whose object code is excellent. Unfortunately, the compiler writer has to design a new semantic algebra for each source language, and must manually implement the translation from this semantic algebra to the target machine code. We would classify high-level semantics as a compiler description language, rather than as a true semantic meta-language.

To conclude, action semantics has opened up a rich field of ideas to be explored. We have been able to exploit the structure of action semantics to build a working (but primitive) compiler generator, in a comparatively short time. We intend to exploit this structure further to analyse source languages in ways never before attempted in any compiler generator. In short, we believe that a *realistic* semantics directed compiler generator is now feasible.

References

1. D. F. Brown. Sort inference in action semantics. Research report, Department of Computing Science, University of Glasgow, Scotland, 1992. In preparation.
2. P. Lee. *Realistic Compiler Generation.* MIT Press, Cambridge, Massachusetts, 1989.
3. P. D. Mosses. SIS – semantics implementation system, reference manual and user guide. Departmental Report DAIMI MD–30, Computer Science Department, Aarhus University, Denmark, 1979.
4. P. D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, England, 1992. In preparation.
5. P. D. Mosses and D. A. Watt. The use of action semantics. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 135–163. North Holland, Amsterdam, Netherlands, 1987.
6. H. Moura. An implementation of action semantics. Research report, Department of Computing Science, University of Glasgow, Scotland, 1992. In preparation.
7. L. Paulson. *A compiler generator for semantic grammars.* PhD thesis, Stanford University, California, 1981.
8. D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.
9. D. A. Schmidt. Detecting stack-based environments in denotational definitions. *Science of Computer Programming*, 11:107–131, 1988.
10. D. A. Schmidt and S. Even. Type inference for action semantics. In N. Jones, editor, *ESOP '90, 3rd European Symposium on Programming*, pages 118–133, Copenhagen, Denmark, 1990. Springer-Verlag, Berlin, Germany. Lecture Notes in Computer Science, Volume 432.
11. M. Wand. A semantic prototyping system. *SIGPLAN Notices (SIGPLAN '84 Symp. On Compiler Construction)*, 19(6):213–221, June 1984.
12. D. A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, England, 1991.

A The Action Semantics of MINI-△

MINI-△ is a simple imperative language. Its main features are integers, truth values, assignment, if- and while-commands, procedures and parameters. The following semantic description of MINI-△ is precisely the input for the ACTRESS system (although parts of it have been omitted due to space constraints).

```
1   Commands

(1)     execute _ :: Command -> act .

(2)     execute [[ ASSIGN I:Identifier E:Expression ]] =
                evaluate E then store the value in the cell bound to id I .

(3)     execute [[ CALL I:Identifier A:Argument ]] =
                giveArgument A then
                enact (the abstraction bound to id I with the argument) .

(4)     execute [[ CSEQ C1:Command C2:Command ]] =
                execute C1 and then execute C2 .

(5)     execute [[ LET D:Declaration C:Command ]] =
                furthermore elaborate D
                hence execute C .
```

```
(6)     execute [[ IF E:Expression C1:Command C2:Command ]] =
                evaluate E then
                | execute C1 else execute C2 .

(7)     execute [[ WHILE E:Expression C:Command ]] =
                unfolding
                | evaluate E then
                | | | execute C and then unfold
                | | else complete .

2   Expressions

(1)     evaluate _ :: Expression -> act .

(2)     evaluate [[ INT I:Integer ]] = give integerValuation I .

(3)     evaluate [[ BOOL B:Boolean ]] = give booleanValuation B .

(4)     evaluate [[ ID I:Identifier ]] =
                give the value bound to id I or
                give the value stored in the cell bound to id I .

(5)     evaluate [[UNARY O:Operator E:Expression ]] =
                evaluate E then apply O .

(6)     evaluate [[BINARY E1:Expression O:Operator E2:Expression ]] =
                | | evaluate E1 then give the value label #1
                | and
                | | evaluate E2 then give the value label #2
                then apply O .

(7)     apply _ :: Operator -> act .

(8)     apply [[ ADD ]] = give sum(the value#1, the value#2) .

...

(15)    apply [[ NOT ]] = give not(the value) .

3   Declarations

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

(2)     elaborate [[ CONST I:Identifier E:Expression ]] =
                evaluate E then bind id I to the value .

(3)     elaborate [[ VAR I:Identifier T:Type ]] =
                allocateForType T then bind id I to the cell .

(4)     elaborate [[ PROC I:Identifier F:FormalParameter C:Command ]] =
                bind id I to closure abstraction
                | furthermore bindParameter F
                | hence execute C .

(5)     elaborate [[ DSEQ D1:Declaration D2:Declaration ]] =
                elaborate D1 before elaborate D2 .

(6)     allocateForType _ :: Type -> act .

(7)     allocateForType [[ INTTYPE ]] = allocate an [integer] cell .

(8)     allocateForType [[ BOOLTYPE ]] = allocate a [truth-value] cell .
```

B The Generated MINI-△ Compiler

Consider the MINI-△ source program in Figure 3. Applying the parser, actioneer, and sort checker to this source program, we obtain the program action shown in Figure 4. Applying the code generator to this action, we obtain the object code shown in Figure 5. Figure 6 shows the improved object code we would obtain after performing the transformations discussed in Section 6.

```
let
  const n ~ 1000000;
  var x : Integer
in
  begin
    x := n;
    while x > 0 do x := x - 1
  end
```

Fig. 3. A MINI-△ loop program

```
|furthermore
|||give 1000000 then bind "n" to the value
||before
|||allocate an [integer]cell then bind "x" to the cell
hence
||give the value bound to "n" then store the value in the cell bound to "x"
|and then
||unfolding
||||||give the value stored in the cell bound to "x"
||||||then give the value label #1
|||||and
||||||give 0 then give the value label #2
||||then give isGreaterThan(the value#1,the value#2)
|||then
|||||||||give the value stored in the cell bound to "x"
|||||||||then give the value label #1
||||||||and
|||||||||give 1 then give the value label #2
|||||||then give difference(the value#1,the value#2)
||||||then store the value in the cell bound to "x"
|||||and then unfold
||||else complete
```

Fig. 4. The program action for the loop program

```
#include "runtime.c"
DATUM _d1,_d2,_d3,_d4;
BINDINGS _b1,_b2;
int main () {
    _d1 = _MAKE_INTEGER (1000000); _b1 = _BIND ("n",_d1);
    _d1 = _ALLOCATE_A_CELL (); _b2 = _BIND ("x",_d1);
    _b1 = _OVERLAY_BINDINGS (_b1,_b2);
    _d1 = _BOUND ("n",_b1); storage[(_BOUND ("x",_b1)).datum.cell] = _d1;
_repeat_1:
    _d1 = storage[_BOUND ("x",_b1).datum.cell];
    _d2 = _MAKE_INTEGER (0); _d3 = _IS_GREATER_THAN(_d1, _d2);
    if (_d3.datum.truth_value == true) {
        _d1 = storage[_BOUND ("x",_b1).datum.cell];
        _d2 = _MAKE_INTEGER (1); _d4 = _DIFFERENCE(_d1, _d2);
        storage[(_BOUND ("x",_b1)).datum.cell] = _d4; goto _repeat_1; }
    else { ; };
    exit (0);
_failure_0:
    exit (1);
}
```

Fig. 5. Object code for the loop program

```
#include "runtime-notags.c"
DATUM _d1;
int main () {
    storage[0].integer = 1000000;
_repeat_1:
    _d1.truth_value = storage[0].integer > 0;
    if (_d1.truth_value) {
        storage[0].integer = storage[0].integer - 1;
        goto _repeat_1; }
    else { ; };
    exit (0);
_failure_0:
    exit (1);
}
```

Fig. 6. Improved object code for the loop program