Generating Action Compilers by Partial Evaluation

Anders Bondorf
DIKU, Department of Computer Science
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
Internet: anders@diku.dk

Jens Palsberg
Computer Science Department
Aarhus University
DK-8000 Aarhus C, Denmark
Internet: palsberg@daimi.aau.dk

Abstract

Compiler generation based on Mosses' action semantics has been studied by Brown, Moura, and Watt, and also by the second author. The core of each of their systems is a handwritten action compiler, producing either C or machine code.

We have obtained an action compiler in a much simpler way: by partial evaluation of an action interpreter. Even though our compiler produces Scheme code, the code runs as fast as that produced by the previous action compilers.

1 Introduction

Action semantics is a framework for formal semantics of programming languages, developed by Mosses [16, 17, 18] and Watt [19, 26]. It differs from denotational semantics in using semantic entities called *actions*, rather than higher-order functions.

Compiler generation based on action semantics has been studied by Brown, Moura, and Watt [6], and also by the second author [22, 20, 21].

Journal of Functional Programming, 6(2):269–298, 1996. Also in Proc. FPCA'93, pages 308–317.

The core of each of their two action semantics directed compiler generators is a handwritten action compiler, producing either C or machine code. These compilers are rather complicated and, due to their complexity, difficult to modify.

We have obtained an action compiler in a much simpler way: by partial evaluation of an action interpreter. The action interpreter is written in Scheme, and is straightforward, except for some binding time improving parts [10]. We have obtained the action compiler using the Similix partial evaluator [4, 5, 2, 3]. Even though our action compiler produces Scheme code, the code runs as fast as that produced by the previous action compilers.

We have used the generated action compiler in an action semantics directed compiler generator. The generated compilers produce target code that, by comparison with measurements reported in [13], is at least ten times faster than that produced by the compilers generated by the classical systems of Mosses [15], Paulson, [23], and Wand [24], but still around 100 times slower than target programs produced by handwritten compilers.

None the classical systems of Mosses, Paulson, and Wand include a binding-time analysis. Binding-time analysis enables computations to take place at compile-time and it is an integrated component of the Similix partial evaluator. This is part of the reason why our system is faster than the classical systems.

The claim that partial evaluation may lead to the generation of acceptable compilers has been made many times, for example in the first paper on Jones, Sestoft, and Søndergaard's Mix partial evaluator [11]. Jørgensen [12] used partial evaluation to generate a compiler for a lazy functional language, and this compiler emits code that compares favorably to that emitted by handwritten compilers. Consel and Danvy [7] used partial evaluation to generate a compiler from denotational semantics, and their compiler produces target code that is only two times slower than that produced by handwritten compilers.

A key point in the development of Consel and Danvy [7] is to identify and process the static semantics by partial evaluation. Our approach is similar: we identify and process the static semantics partly by a separate action type checker and partly by partial evaluation. The observation that our system leads to around 50 times slower target code than the system of Consel and Danvy indicates, in our opinion, that more powerful analyses might take place in the separate action type checker. We leave investigations of this to future work.

In the following section we explain the principles of action semantics

and in Section 3 we outline the problems connected to compiling actions. In Section 4 we discuss our action interpreter, in Section 5 we focus on its binding time improving parts, and in Section 6 we give a performance evaluation. Finally, in Section 7 we conclude and outline directions for further work.

2 Action Semantics

Actions reflect the gradual, stepwise nature of computation. A performance of an action, which may be part of an enclosing action, either

- *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally);
- *escapes*, corresponding to exceptional termination (the enclosing action is skipped until the escape is trapped);
- *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- *diverges*, corresponding to nontermination (the enclosing action also diverges).

The performance of an action receives and produces *transients* (tuples of data, used to hold intermediate results); it receives and produces *bindings* of tokens to data (environments), and it manipulates an implicit, single-threaded *store*. Actions may also *communicate*, but here we consider only single-agent performance where communication is uninteresting. Actions themselves are not data, but they can be incorporated in so-called abstractions, which are data, and subsequently 'enacted' back into actions.

Transients are produced only on completion or escape, and bindings are produced only on completion. In contrast, changes to the store are made *during* action performance, and are unaffected by subsequent divergence or failure.

Dependent data are entities that can be *evaluated* to yield data during action performance. (In [18], dependent data are called yielders; they may be thought of as expressions.) The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the store. Evaluation cannot affect the current information. Data is a special case of dependent data, and it always yields itself when evaluated.

The language of actions is called action notation. We use a subset of action notation which was also studied in [22, 20] and defined in [21]. This subset covers roughly half of the full action notation and is sufficiently general to allow the easy description of Lee's HypoPL [22] and a non-trivial subset of Ada [20]. For an example of an action semantic description using this subset, see Appendix C. Scaling up our results to full action notation would require significant extension of our action interpreter, especially to handle communication.

The meaning of the action notation used in Appendix C is informally presented in the following three tables. The first table describes five primitive actions. The symbols D, D_1, D_2 stand for dependent data.

Primitive action:	Informal meaning:
give D	Creates a piece of transient information.
bind D_1 to D_2	Creates a binding.
store D_1 in D_2	Changes the store.
allocate integer cell	Allocates a fresh cell in the store.
check D	If D evaluates to the value true, then check D completes, otherwise it fails.

The second table describes seven pieces of dependent data. The symbol p stands for a positive integer and the symbol T stands for a token.

Dependent data:	Informal meaning:
the given D #p	Yields the p'th component of sort D of the received transients.
it	Yields the first and only component of the received transients.
the D bound to T	Yields the datum of sort D to which T is bound by the received bindings.
the D_1 stored in D_2	Yields the data of sort D_1 stored in the cell yielded by D_2.
sum(D_1,D_2)	Integer addition.
D_1 is less than D_2	Integer comparison.
not D	Boolean negation.

The third table describes five binary action combinators. The symbols A_1, A_2 stand for actions. Common for the first four is that A_2 is only performed if A_1 completes. In contrast, for A_1 or A_2, the action A_2 is only performed if A_1 fails.

Action combinator:	Informal meaning:
A_1 then A_2	Passes on transients from A_1 to A_2.
A_1 and then A_2	Combines the transients produced by A_1 and A_2.
A_1 before A_2	Accumulates the bindings produced by A_1 and A_2.
furthermore A_1 hence A_2	Lets A_2 receive the bindings produced by A_1.
A_1 or A_2	If A_1 fails, then A_2 is performed.

In the chosen subset of action notation, just one action combinator gives the possibility of divergence. That combinator is unfolding A, where A is an action. unfolding A represents the (in general, infinite) action formed by continually substituting A for the primitive action unfold. Our subset of action notation restricts the possible actions A in unfolding A to be those where unfold occurs exactly once and in a tail recursive position.

For a precise definition of the above notation, see Appendix B.

3 Action Compilation

To obtain an action semantics directed compiler generator, an action compiler is required. Such a compiler can then, as usual, be composed with a preprocessor that, given a language definition, expands programs to actions. This expansion is straightforward because action semantics is compositional.

It is non-trivial to compile actions into efficient code. For example, the primitive action "`bind_to_`" is used both in the semantics of constants (for example "`bind` "max" `to` 100"), in the semantics of variables (for example "`bind` i `to the given integer-cell` ..."), and in the semantics of procedures (for example "`bind` "fact" `to closure abstraction` ..."). In each of the examples, the action produces a binding, but no transients, and it does not modify the storage. It is a task of the implementer of actions to discover that these three binding actions can be treated differently and use that knowledge to produce efficient code. The following section describes our approach which is a combination of static type checking and partial evaluation.

As another example, consider the binary action combinator "`before`". It is used in the semantics of declaration sequences (for example, "`(bind` "max" `to` 100`) before (bind` "i" `to the given integer-cell ...)`"). The example action produces two bindings. In general, A_1 `before` A_2 produces the bindings produced by A_1 overlayed by those produced by A_2. Moreover, it gives the transients given by A_1 concatenated with those given by A_2.

It is a task of the implementer to discover that the example action does not produce any transients and use that to avoid generating superfluous code. Instead appropriate information must at compile-time be propagated to later actions. The following section demonstrates how this work can be divided by combining static type checking and partial evaluation.

The two existing action compilers are handwritten; we automatically generate one from an action interpreter. The action compiler is obtained by applying the (self-application generated) compiler generator of Similix 5.0 to an action interpreter. This approach to compiler generation can be informally summarized as follows. Both the action interpreter `int` and the partial evaluator `Similix` are written in Scheme. The following equation expresses that when we execute `int` on the action `act` together with `input`, then we get `output`. (The notation `<d1 d2>` means a tuple of `d1` and `d2`.)

```
Scheme int <act input> = output
```

The next equation is the so-called mix-equation. It expresses that instead of running a program on all its input, we get the same result by first executing the partial evaluator on the program and part of its input, and then executing the resulting program with the remaining input.

```
Scheme prg <d1 d2> = Scheme (Scheme Similix <prg d1>) d2
```

The self-application generated compiler generator, `cogen`, of Similix can be expressed as follows.

```
cogen = Scheme Similix <Similix Similix>
```

The action compiler `comp` is then defined by

```
comp = Scheme cogen int
```

To see that `comp` is indeed an action compiler, notice that if `act` is an action, and we define

```
code = Scheme comp act
```

then

```
code
= Scheme comp act
= Scheme (Scheme cogen int) act
= Scheme (Scheme (Scheme Similix <Similix Similix>) int) act
= Scheme (Scheme Similix <Similix int>) act
= Scheme Similix <int act>
```

so

```
Scheme code input
= Scheme (Scheme Similix <int act>) input
= Scheme int <act input>
= output
```

We have used the generated action compiler in an action semantics directed compiler generator. This system was essentially obtained by replacing the hand-written action compiler in the Cantor system of the second author [22, 20, 21] with the automatically generated action compiler. Thus, a compiler generated by our system first expands the input program to an action, then it type checks the action, and finally it runs the action compiler on the result, see Appendix D for an example.

4 The Action Interpreter

Our subset of action notation, see Appendix A, has an operational semantics [21]. From that we have systematically derived an action interpreter, written in Scheme, see Appendix B. Its size is approximately half the size of the operational semantics. In previous work [21], the second author defined and proved the correctness of a type analysis and a code generator for our subset of action notion. In comparison, our interpreter is less than a third of the size of the code generator, and about one tenth of the size of the (sketchy) proof.

It is possible to consider our action interpreter as an alternative semantics of our subset of action notation (although not equally useful for all purposes). If we do so, then our approach has the advantage of requiring neither a code generator nor a correctness proof. This is because our system generates a compiler directly from a (rather short) semantics of actions. Correctness concerns are moved one level "down": is the partial evaluator correct? This is a major advantage since such a correctness proof must be

carried out only once, irrespectively of the application of the partial evaluator. For examples of such proofs for toy partial evaluators, see the papers by Gomard [9] and Wand [25]. Note, however, that Similix has *not* been proved correct.

Our subset of action notation is statically typed. We use the *same* type-checker as the second author did in his previous work [21, 22, 20]. Thus, we first run the type-checker and then the interpreter. Both operate on the same abstract syntax: the type-checker inserts various information in the syntax-tree. This information is about type correctness, tokens, and number of transients, see later.

We have experimented with merging the type-checker with the interpreter. This complicates the program structure considerably, however, and creates the problem of ensuring that all type-checking code depends only on static data. Since the type system is indeed static and since type-checking produces only a moderate amount of information, we decided to run the type-checker first and then let the interpreter exploit the type information.

Our subset of action notation is statically scoped and, like the whole of action notation, single-threaded. The latter means that just one store is sufficient to execute actions. Single-threadedness is an issue in connection with the "`or`" combinator. Consider for example the action `act1 or act2`. The two actions `act1` and `act2` are alternatives of which just one should be performed. If the chosen action fails, however, then the other should be tried on the *same* store on which the performance of the first chosen action began ("back-tracking"). The semantics of actions defines that if the first chosen action has changed the store or in some other way "committed" to the current alternative (like `cut` in Prolog), then back-tracking is not allowed, so the performance of the entire action fails. This ensures single-threadedness.

Our action interpreter is called `ev-act` and has the following structure:

```
(define (ev-act act dats env commit c e f)
   ...)
```

Here, `act` is the action to be interpreted; `dats` is a tuple that represents the transients received by `act`; `env` is a map that represents the bindings received by `act`; `commit` is a boolean that tells if the current action has committed; and `c`, `e`, and `f` are continuations, to be used should the performance complete, escape, or fail, respectively. Since an action on completion may produce both transients and bindings, the complete-continuation has the form:

```
(lambda (dats vs commit) ...)
```

The `vs` argument is a list of values that the continuation will use to extend its environment, see later. On escape, an action can only produce transients, so the escape-continuation does not have the `vs` argument. A failing action produces neither transients nor bindings, so the fail-continuation has only a `commit` argument.

Notice that there is no store argument; we represent the store by a globalized Scheme vector. This directly reflects the intention with the store in action notation: it is implicit and by definition single-threaded. The store is *not* structured as a stack because storage can be allocated at any time during action performance.

Consider the following excerpt of the action interpreter. It is the part defining the "`bind_to_`" action:

```
(define (ev-act act dats env commit c e f)
  (casematch act
    ...
    (('bind_to_ token dep type-correct)
     (if (error? type-correct)
         (f commit)
         (ev-dependent
          dep dats env
          (lambda (dat) (c (0dats) (list dat) commit))
          (failure f commit))))
    ...))
```

The abstract syntax of the bind action is `('bind_to_ token dep type-correct)`. The `type-correct` field is inserted by the action type-checker. This field is `error` if `dep` is not type-correct, and in this case the interpreter correctly uses the `f` continuation (passing on the `commit` value). If `dep` is type-correct, then the interpreter proceeds by evaluating `dep`. This will either yield a datum ("`dat`") or lead to an error (for example at an attempt to compute the head of an empty list).

Notice the application `(0dats)` of the nullary constructor `0dats` in the above piece of Scheme code: it reflects that the `bind` action gives no transients. Notice also that the second argument to `c` is a list (with one element, namely `dat`), *not* a binding of `token` to `dat`. This is possible since our subset of action notation is statically scoped. Intuitively, tokens may only be "synthesized" in ways that can be understood statically. The information that

some particular token is being bound is propagated to later actions by the action type-checker. This means that we can write the interpreter so that it ignores the `token` field in `('bind_to_ token dep type-correct)`. It is sufficient to pass the result of evaluating "`dep`" to the continuation. This way of writing the interpreter considerably improves the binding times.

In general, the second argument to `c` (a list of values) will flow to some continuation which then extends its environments using this list, see the treatment of `before` below. This may be understood as a generalization of binding in a call-by-value lambda calculus, where evaluation of $(\lambda x.E)(E')$ proceeds by first evaluating E' to some value v, and then extending the environment by binding x to v, and finally evaluating E.

If we drop the restriction that actions should be statically scoped, then the interpreter will manipulate tokens in ways that cannot be understood statically. Experience with an early version of our action interpreter indicates that if static scoping is not assumed, then the target code may run five times slower.

Consider then the following excerpt of the action interpreter. It is the part defining the `before` combinator:

```
(define (ev-act act dats env commit c e f)
  (casematch act
    ...
    (('infix-op op act1 act2)
     (casematch op
       ...
       (('before tokens1 dl1 dl2)
        (ev-act
         act1 dats env commit
         (lambda (dats1 vs1 commit1)
           (let ((env1 (make-env tokens1 vs1)))
             (ev-act
              act2 dats (env-overlay env1 env) commit1
              (lambda (dats2 vs2 commit2)
                (c (dats-append dats1 dats2 dl1 dl2)
                   (append vs2 vs1) commit2))
              e f)))
         e f))
       ...))...))
```

The abstract syntax of the combination of two actions `act1` and `act2` us-

ing the binary combination `before` is `('infix-op ('before tokens1 dl1 dl2) act1 act2)`. The `tokens1`, `dl1` and `dl2` fields are inserted by the action type-checker. The `tokens1` field is a list of tokens that matches the list of data `vs1` that will be given to the complete continuation of `act1`. This makes it possible to make the extension of the environment mentioned above: first the extension-part is constructed as `(make-env tokens1 vs1)`, and the extension is performed by `(env-overlay env1 env)`.

The `dl1` and `dl2` fields are integers that give the number of transients given on completion by `act1` and `act2`. The operation `dats-append` is controlled by these (static) integers.

In an early version of our action interpreter, the arguments of `ev-act` had the following binding times: `act` was static, `dats` was dynamic, `env` was partially static (tokens static, values dynamic), and `commit`, `c`, `e`, and `f` were all dynamic. These binding times correspond exactly to those implicitly used by the handwritten action compiler of the second author [21]. We partially evaluated that version of the interpreter with respect to some actions and found that the target code ran several times slower than the target programs produced by the handwritten action compiler.

In the current version of the action interpreter, also the `dats` argument is partially static, and, more significantly, the `c` continuation is static. The following section sketches how we obtained the static `c` and why it is significant.

5 Binding Time Improvements

Suppose we are given a piece of straightline code in some programming language and suppose we expand it to an action. Every subaction of this action will complete, so the code obtained by specializing the action interpreter with respect to the action need not build and apply continuations for each subaction it implements. If the complete continuations `c` in the interpreter are dynamic, however, the target program will indeed do just that. As a consequence, every control transfer becomes costly: it requires building a continuation and calling a function (as when running the interpreter).

To obtain reasonable code, we have performed some binding time improvements of the interpreter that make the complete-continuations static. The trouble points are the higher order control transfers where the target code generated by partial evaluation becomes parameterized over the complete-continuation. The improvements involve inserting *eta-redexes* at

appropriate places [4]; these places are marked with comments `bt-imp 3` and `bt-imp 9` in Appendix B. With these improvements, the complete-continuations will be static everywhere, except at higher order control transfers; the target program will consequently only manipulate continuations if there are higher order control transfers.

To obtain an intuitive understanding of why the insertion of eta-redexes improves binding-times, let us consider the two trouble points in turn. First, at `bt-imp 3` we find the value `dat` being applied to some arguments:

```
(dat (ps->d dats1 dl-in)
     commit
     (cl->d c dl-c)       ; bt-imp 3
     e f))
```

The operation `cl->d` creates an eta-redex around the complete-continuation `c`. (The extra parameter `dl-c` is present for other reasons; this is explained below.) Essentially, `cl->d` is of the form

```
(lambda (x1 x2 x3) (c x1 x2 x3))
```

The idea behind the eta-redex is the following. The value `dat` is a function which according to the binding-time analysis is not statically known. The binding analysis will therefore make all arguments to `dat` dynamic. Had no eta-redex been inserted, `c` would have been made dynamic, even though we want it to be static. Now, the eta-redex serves as "padding" around `c` so that `c` stays static while the whole eta-redex becomes dynamic. Intuitively, this works because it makes sense to let the static `c` be applied to the dynamic arguments `x1`, `x2`, and `x3`.

Next, at `bt-imp 9` we have

```
(c (lambda (dats commit c e f)
     (ev-act act (d->ps dats dl-in)
             env commit
             (d->cl c dl-c)         ; bt-imp 9
             e f))))
```

The operation `d->cl` creates an eta-redex around the complete-continuation `c`.

Like `cl->d`, `d->cl` is essentially of the form

```
(lambda (x1 x2 x3) (c x1 x2 x3))
```

Here, `c` is not statically known. It is used as the complete-continuation argument to `ev-act`, however, so we need it to be static; otherwise, the binding-time of all complete continuations will be dynamic. Again, the eta-redex serves as "padding" around `c` so that `c` stays dynamic while the whole eta-redex can be treated as being static. Intuitively, this works because it makes sense to have a static continuation where its arguments `x1`, `x2`, `x3` are dynamic.

For a thorough treatment of the insertion of eta-redexes, see [8]. In the terminology of that paper, the trouble point `bt-imp 3` is an occurrence of "a static value in a dynamic context" and the trouble point `bt-imp 9` is an occurrence of "a dynamic value in a static context".

However, care must be taken now that the complete-continuations are static. The reason is that the Similix specializer now "believes" that it can statically compute every complete-continuation as long as there are no higher order control transfers. This is of course false in the presence of loops (`unfolding`): the specializer will loop (analogue: a compiler that tries to build run-time stacks at compile-time). The cure is to insert an additional binding time operation `collapse` at loops; see the comment `bt-imp 4` in the interpreter text. This operation locally generates a dynamic complete-continuation `c1`. The effect is that applications of the complete-continuations that represent "iterate loop" (`unfold`) are not beta-reduced at partial evaluation time, so the target code will be parameterized over `c1`.

To see how `collapse` works, notice that it contains the code

```
(if #t c1 (generalize c1))
```

Clearly, this evaluates to `c1`. The binding-time analysis, however, will notice that the operation `(generalize c1)` enforces `c1` to be dynamic.

The `collapse`-operation is also used to avoid code duplication even when termination is guaranteed; see `bt-imp 5`, `7`, `10`, and `12`. Currently, no automatic method for inserting `collapse`-operations exists.

The binding time improvements `bt-imp 1`, `2`, `6`, `8`, and `11` ensure that the transients `dats` become partially static everywhere, except when being passed to dynamic continuations. To do this, the number of transients is required, and this is provided by the action type checker as the value called `dl` (or `dl-in`, `dl-c`, etc.). The effect is that the target code will pack and unpack transients only when being passed to continuations; in

target straightline code, each transient will be represented by its own Scheme variable. These binding-time improvements are the data structure analogues of the ones for functions (`bt-imp 3` and `bt-imp 9`) that were explained above (see also [8]).

The operation `_sim-memoize` (just above `bt-imp 4`) is also a kind of binding time improvement: it forces the partial evaluator to specialize/memoize at this point (instead of using the default "dynamic conditional" strategy [5]). This results in shorter and somewhat faster target code (with fewer function calls), and also faster partial evaluation. Current work by Malmkjær addresses finding (good) specialization points automatically [14].

We have experimented with making the escape and fail continuations static; this yields bigger and/or slower target programs. It is not surprising that target programs become bigger since escape and fail are the exceptional outcomes of performing an action: having three static continuations corresponds to wanting to have three different pieces of code at the same place in the program. Target programs can be made smaller by (extensive) use of `collapse`, but this slows them down since `collapse` yields target code that unpacks and packs transients. Our conclusion is "optimize the straightline code".

6 Performance Evaluation

We have tested our compiler generator on specifications of Lee's HypoPL and a substantial subset of Ada. These language specifications may be found in [22, 20, 21].

Our example programs are a bubblesort program (written in both HypoPL and Ada), and programs for performing the sieve of Erathosthenes and the algorithm of Euclid (written in Ada). These programs may be found in [21].

The four example programs were all compiled both by compilers generated by the Cantor system of the second author, and by compilers generated by applying the (self-application generated) compiler generator of Similix 5.0 to the interpreter in Appendix B.

In the following tables we have listed some timings obtained on a SPARC 1 running Scm version 4b4. The tables also show (in the fourth column) the timings after we have discounted the overhead imposed by interpreting Scheme programs by Scm rather than compiling them by a Scheme compiler. After comparing Scm with both MIT-Scheme, Scheme–>C, and Chez

Scheme, we (rather conservatively) estimate this factor to be 5. For comparison, the tables also show timings of the Cantor system running on a SPARC 1. All timings are in seconds.

The three tables show the times taken to generate compilers, to compile the example programs, and to run the target programs, respectively. The last two tables also show (in the last column) how many times faster the compilers and target programs of our system were (with the factor 5 counted).

Compiler-gen. times	Cantor	Ours	Ours/5
HypoPL	3	179	36
Mini-Ada	9	185	37

Compile times	Cantor	Ours	Ours/5	Speed-up
bubble.hpl	486	56.6	11	43
bubble.ad	542	40.9	8.2	66
sieve.ad	377	34.7	6.9	54
euclid.ad	136	20.5	4.1	33

Run times	Cantor	Ours	Ours/5	Speed-up
bubble.hpl	0.1	0.13	0.026	3.8
bubble.ad	0.9	6.0	1.2	0.75
sieve.ad	1.2	2.9	0.58	2.1
euclid.ad	0.8	3.0	0.60	1.3

The first two tables indicate that our system, in contrast to the Cantor system, yields relatively long compiler-generation times and relatively short compile times, rather than the opposite. This makes our new system much better for experimental use than the Cantor system.

The run times in the third table are encouraging, considering that the action compiler in the Cantor system is designed specifically to generate SPARC code [21]. In contrast, our action compiler generates Scheme code.

Currently, we do not understand why there is a difference between the HypoPL and the Ada bubblesort programs. However, this difference seems to depend on the machine the tests are run on: on an HP9000s730, the ratio between our run-times for "bubble.ad" and "bubble.hpl" (6.0/0.13) is 40% smaller.

It is apparent from the target programs that a reasonable amount of constant folding has been performed by the partial evaluator: it has not just compiled from actions into Scheme. Being an offline partial evaluator,

one should not *a priori* believe that Similix would do such constant folding. That this nevertheless happens is due to the postprocessing phase of Similix.

Even more constant folding can be obtained by partially evaluating the target program (with no static input) once more. By doing this for the bubble.hpl target program, the program becomes twice as fast. That is, the speed-up compared to Cantor can be improved to around 8 from the 3.8 above. However, the time for performing this second specialization is significant, about ten times larger than the current compile time (56.6 seconds) listed above. For a small example of partially evaluating a target program a second time, see Appendix D.

A worthy experiment would be to disable postprocessing in the first partial evaluation and then to do partial evaluation a second time. This would enable a clear separation between the compilation from actions into Scheme (first partial evaluation), and the extra constant folding (second partial evaluation).

The target programs obtained using the Cantor system and the system of Brown, Moura, and Watt are about 100 times slower than those emitted by hand-written compilers. With the above measurements, our system yields roughly the same results. Note that the system of Brown, Moura, and Watt does *not* distinguish between committed and non-committed failures [27]. We believe that this is a significant simplification because our target programs contain a considerable amount of code to distinguish failures.

7 Conclusion

We have obtained an action compiler by partially evaluating an action interpreter. We have used the automatically generated action compiler in an action semantics directed compiler generator, and found that it yields faster compilers and as fast target programs as the previous Cantor system. There is still room for improvement, however: compared to C, our target code is still almost 100 times slower.

Improvements of the compiler can be obtained by performing static analyses of actions and exploiting the information in the interpreter. The current action interpreter only takes advantage of the information provided by the action type-checker that came for free with the subset of action notation we have considered. One specific idea for improvement is to split the environment into two parts: one where the bound data are known to be static, and another for the remaining bindings. The work needed to perform compile-

time analyses of actions is of course independent of whether actions are eventually compiled by a hand-written compiler or by a compiler generated by partial evaluation. However, we believe that it is easier to rewrite the action interpreter to take advantage of additional information generated by static analyses than it is to rewrite a hand-written compiler.

It may hinder practical use of our system that target programs are in Scheme, which is rather slow compared to C. It might be worthwhile rewriting the action interpreter in C, and then use Andersen's partial evaluator of C programs [1].

At the initial stages of our project we considered writing a *meta-interpreter* for action semantic descriptions. The arguments of such a meta-interpreter should be both a language semantics and a program in that language. Informally, we might have a meta-interpreter `meta-int` so that for a semantics `sem` we get

```
Scheme meta-int <sem prg input> = output
```

We can now get a meta-compiler, `meta-comp`, by defining

```
meta-comp = Scheme cogen meta-int
```

It could then be possible to generate a compiler by applying the meta-compiler to a particular language semantics. Informally, we might define

```
new-comp = Scheme meta-comp sem
```

The generated `new-comp` will compile programs in the defined language into Scheme. This approach does not seem worthwhile, however, because the efficient implementation of actions requires extensive type analysis. The result of this analysis is most naturally put in the syntax tree of the analyzed action, but using the meta-interpreter approach, this action is never generated! It might of course be possible to recompute the type information whenever needed, and then hope that a partial evaluator could perform the necessary caching. The key point would be to keep the recomputation under static control. Unfortunately, with the Similix 5.0 system, this is impossible.

8 Acknowledgments

This work has been supported in part by the Danish Research Council under the DART Project (5.21.08.03). The authors thank Peter Mosses, Michael

Schwartzbach, and the anonymous referees for helpful comments on a draft of the paper.

A Abstract Syntax of Actions

This appendix presents the syntax of actions that is processed both by the action type-checker and the action interpreter. The action type-checker updates the fields denoted `Type-correct`, `Dats-length`, and `Tokens`.

```
Act  ::= complete | (escape Dats-length) | fail
    | commit | diverge | regive
    | (give Dependent Type-correct)
    | (check Dependent Type-correct)
    | (bind_to_ Token Dependent Type-correct)
    | (store_in_ Dependent Dependent Type-correct)
    | allocate-truth-value-cell
    | allocate-integer-cell
    | (batch-send Dependent Type-correct)
    | batch-receive-an-integer
    | (enact-application_to_ Dependent Tuple
        Type-correct Dats-length Dats-length)
    | (indivisibly Act)
    | (unfolding Unfolding Dats-length)
    | (infix-op Act-Infix Act Act)

Tuple  ::= empty-tuple | (dependent Dependent)
    | (comma Tuple Tuple Dats-length Dats-length)
    | them

Dependent  ::= true | false | (nat Natural)
    | (empty-list-&_-list Type)
    | (closure-abstraction-of_&-perhaps-using_-act
        Act Data Dats-length Dats-length)
    | (unary-op Unary Dependent)
    | (binary-op Binary Dependent Dependent)
    | it
    | (the-given_#_ Datum Natural Dats-length)
    | (the_bound-to_ Datum Token)
    | (the_stored-in_ Datum Dependent)

Unfolding  ::= (infix-op Infix Act Unfolding) | (unfold Dats-length)

- - - - - - - - - - - - - - - - - - - - - - - -

Act-Infix ::= Infix
    | (furthermore-hence Tokens Dats-length Dats-length)
    | (furthermore-thence Tokens)
```

```
Infix  ::= (and-then Dats-length Dats-length)
    | then
    | (before Tokens Dats-length Dats-length)
    | (trap Dats-length Dats-length)
    | (or Dats-length)

Unary  ::= not | negation | list-of | head | tail

Binary ::= both | either | sum | difference
    | concatenation | is | is-less-than
    | component#_items_

-----------------------------------------

Datum        ::= datum | cell | abstraction | list
               | (datum-or Datum Datum) | (type Type)
Data         ::= empty-data | (type Type)
               | (comma Data Data)
Type         ::= truth-value | integer
               | truth-value-cell | integer-cell
               | (_-list Type)

Type-correct ::= ok | error
Dats-length  ::= Natural
Tokens       ::= (V*)
Token        ::= V
V            ::= "the set of Scheme symbols"
Natural      ::= "the set of Scheme numbers"
```

B Text of the Action Interpreter

This appendix presents the complete text of our action interpreter.

```
(define (int act in-file out-file)
  (init! in-file out-file)
  (ev-act act (0dats) (init-env) #f
          (lambda (dats vs commit)  (close!) '__completed)
          (lambda (dats commit)  (close!) '__escaped)
          (lambda (ct)  (close!) '__failed)))

(define (ev-act act dats env commit c e f)
  (casematch act
```

```
('complete
 (c (0dats) '() commit))
(('escape dl)
 (e (ps->d dats dl) commit))     ; bt-imp 1
('fail
 (f commit))
('commit
 (c (0dats) '() #t))
('diverge
 (loop))
('regive
 (c dats '() commit))
(('give dep type-correct)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep dats env
      (lambda (dat) (c (1dats dat) '() commit))
      (failure f commit))))
(('check dep type-correct)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep dats env
      (lambda (dat)
        (if (equal? dat #t)
            (c (0dats) '() commit)
            (f commit)))
      (failure f commit))))
(('bind_to_ token dep type-correct)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep dats env
      (lambda (dat) (c (0dats) (list dat) commit))
      (failure f commit))))
(('store_in_ dep1 dep2 type-correct)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep1 dats env
      (lambda (dat1)
        (ev-dependent
         dep2 dats env
```

```
         (lambda (dat2)
           (update-store! dat2 dat1)
           (c (0dats) '() #t))
         (failure f commit)))
      (failure f commit))))
('allocate-truth-value-cell
 (c (1dats (allocate-cell!)) '() #t))
('allocate-integer-cell
 (c (1dats (allocate-cell!)) '() #t))
(('batch-send dep type-correct)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep dats env
      (lambda (dat)
        (output! dat)
        (c (0dats) '() #t))
      (failure f commit))))
('batch-receive-an-integer
 (c (1dats (input!)) '() #t))
(('enact-application_to_
  dep tuple type-correct dl-in dl-c)
 (if (error? type-correct)
     (f commit)
     (ev-dependent
      dep dats env
      (lambda (dat)
        (ev-tuple
         tuple dats env
         (lambda (dats1)
           (dat (ps->d dats1 dl-in)  ; bt-imp 2
                commit
                (cl->d c dl-c)       ; bt-imp 3
                e f))
         (failure f commit)))
      (failure f commit))))
(('indivisibly act)
 (ev-act act dats env commit c e f))
(('unfolding unf dl-c)
 ; force insertion of a specialization point:
 (_sim-memoize
  (ev-unfolding unf act dats env commit
                (collapse c dl-c)    ; bt-imp 4
                e f)))
```

```
((’infix-op op act1 act2)
 (casematch op
   ((’and-then dl1 dl2)
    (ev-act
     act1 dats env commit
     (lambda (dats1 vs1 commit1)
       (ev-act
        act2 dats env commit1
        (lambda (dats2 vs2 commit2)
          (c (dats-append dats1 dats2 dl1 dl2)
             vs2 commit2))
        e f))
     e f))
   (’then
    (ev-act act1 dats env commit
            (lambda (dats1 vs1 commit1)
              (ev-act act2 dats1 env commit1 c e f))
            e f))
   ((’before tokens1 dl1 dl2)
    (ev-act
     act1 dats env commit
     (lambda (dats1 vs1 commit1)
       (let ((env1 (make-env tokens1 vs1)))
         (ev-act
          act2 dats (env-overlay env1 env) commit1
          (lambda (dats2 vs2 commit2)
            (c (dats-append dats1 dats2 dl1 dl2)
               (append vs2 vs1) commit2))
          e f)))
     e f))
   ((’trap dl-c dl-e)
    (let ((c (collapse c dl-c)))     ; bt-imp 5
      (ev-act
       act1 dats env commit c
       (lambda (dats1 commit1)
         (ev-act act2
                 (d->ps dats1 dl-e)  ; bt-imp 6
                 env commit1 c e f))
       f)))
   ((’or dl-c)
    (let ((c (collapse c dl-c)))     ; bt-imp 7
      (ev-act
       act1 dats env #f
       (lambda (dats1 vs1 commit1)
```

```
            (c dats1 vs1 (or commit commit1)))
          (lambda (dats1 commit1) (e dats1 (or commit commit1)))
          (lambda (ct1)
            (if commit1
                (bomb)
                (ev-act act2 dats env commit c e f))))))
       (('furthermore-hence tokens1 dl1 dl2)
        (ev-act
         act1 dats env commit
         (lambda (dats1 vs1 commit1)
           (let ((env1 (make-env tokens1 vs1)))
             (ev-act
              act2 dats (env-overlay env1 env) commit1
              (lambda (dats2 vs2 commit2)
                (c (dats-append dats1 dats2 dl1 dl2)
                   '() commit2))
              e f)))
         e f))
       (('furthermore-thence tokens1)
        (ev-act
         act1 dats env commit
         (lambda (dats1 vs1 commit1)
           (let ((env1 (make-env tokens1 vs1)))
             (ev-act
              act2 dats (env-overlay env1 env) commit1
              (lambda (dats2 vs2 commit2) (c dats2 '() commit2))
              e f)))
         e f))))))

(define (ev-tuple tuple dats env c f0)
  (casematch tuple
    ('empty-tuple
     (c (0dats)))
    (('dependent dep)
     (ev-dependent
      dep dats env (lambda (dat) (c (1dats dat))) f0))
    (('comma tuple1 tuple2 dl1 dl2)
     (ev-tuple
      tuple1 dats env
      (lambda (dats1)
        (ev-tuple
         tuple2 dats env
         (lambda (dats2)
           (c (dats-append dats1 dats2 dl1 dl2)))
```

```
          f0))
       f0))
     ('them
      (c dats))))

(define (ev-dependent dep dats env c f0)
  (casematch dep
    ('true
     (c #t))
    ('false
     (c #f))
    (('nat nat)
     (c nat))
    (('empty-list-&_-list type)
     (c '()))
    (('closure-abstraction-of_&-perhaps-using_-act
      act data dl-in dl-c)
     (c (lambda (dats commit c e f)
          (ev-act act (d->ps dats dl-in) ; bt-imp 8
                  env commit
                  (d->cl c dl-c)         ; bt-imp 9
                  e f))))
    (('unary-op unary dep)
     (ev-dependent dep dats env
             (lambda (dat)
               (casematch unary
                 ('not
                  (c (not dat)))
                 ('negation
                  (c (- 0 dat)))
                 ('list-of
                  (c (list dat)))
                 ('head
                  (if (pair? dat) (c (car dat)) (f0)))
                 ('tail
                  (if (pair? dat) (c (cdr dat)) (f0)))))
             f0))
    (('binary-op binary dep1 dep2)
     (ev-dependent
      dep1 dats env
      (lambda (dat1)
        (ev-dependent
         dep2 dats env
         (lambda (dat2)
```

```
            (casematch binary
              ('both
               (c (and dat1 dat2)))
              ('either
               (c  (or dat1 dat2)))
              ('sum
               (c (+ dat1 dat2)))
              ('difference
               (c (- dat1 dat2)))
              ('concatenation
               (c (append dat1 dat2)))
              ('is
               (c (equal? dat1 dat2)))
              ('is-less-than
               (c (< dat1 dat2)))
              ('component#_items_
               (if (< (length dat2) dat1)
                   (f0)
                   (c (list-ref dat2 (- dat1 1)))))))
          f0))
      f0))
    ('it
     (c (1-1st-dats dats)))
    (('the-given_#_ type-set nat dats-length)
     (c (dats-ref dats nat dats-length)))
    (('the_bound-to_ type-set token)
     (c (lookup-env token env)))
    (('the_stored-in_ type-set dep)
     (ev-dependent
      dep dats env
      (lambda (dat)
        (let ((stored-value (read-store! dat)))
          (if (equal? stored-value "Uninitialized")
              (f0)
              (c stored-value))))
      f0))))

(define (ev-unfolding unf act dats env commit c e f)
  (casematch unf
    (('infix-op op act1 unf1)
     (casematch op
       ('then
        (ev-act
         act1 dats env commit
```

```
         (lambda (dats1 vs1 commit1)
           (ev-unfolding unf1 act dats1 env commit1 c e f))
         e f))
       (('trap dl-c dl-e)
        (let ((c (collapse c dl-c)))     ; bt-imp 10
          (ev-act
           act1 dats env commit c
           (lambda (dats1 commit1)
             (ev-unfolding unf1 act
                     (d->ps dats1 dl-e)  ; bt-imp 11
                     env commit1 c e f))
           f)))
       (('or dl-c)
        (let ((c (collapse c dl-c)))     ; bt-imp 12
          (ev-act
           act1 dats env #f
           (lambda (dats1 vs1 commit1)
             (c dats1 vs1 (or commit commit1)))
           (lambda (dats1 commit1) (e dats1 (or commit commit1)))
           (lambda (ct1)
             (if commit1
                 (bomb)
                 (ev-unfolding unf1 act dats env commit c e f))))))
       (else ; (member op '(and-then before))
        (ev-act act1 dats env commit
                (lambda (dats1 vs1 commit1)
                  (ev-unfolding unf1 act dats env commit1 c e f))
                e f))))
    (('unfold dl-c)
     (ev-act act dats env commit c e f))))

;-------------------------------------------------
; Environments:

(define (init-env) (init-env-cstr))
(define (update-env token dat env)
  (update-env1 (binding-env-cstr token dat) env))
(define (lookup-env token env)
  (let ((binding (car-env-sel env)))
    (if (equal? token (name-env-sel binding))
        (dat-env-sel binding)
        (lookup-env token (cdr-env-sel env)))))
(define (update-env1 binding env)
  (cons-env-cstr binding env))
```

```
(define (env-overlay env2 env1)
  (if (init-env-cstr? env2)
      env1
      (update-env1
       (car-env-sel env2)
       (env-overlay (cdr-env-sel env2) env1))))
(define (make-env tokens vs)
  (let ((arity (length tokens)))
    (let loop ((offset 0))
      (if (equal? offset arity)
          (init-env)
          (update-env (list-ref tokens offset)
                      (list-ref vs offset)
                      (loop (+ offset 1)))))))

;-------------------------------------------------
; Processing dats:

(define (dats-append dats1 dats2 dl1 dl2)
  (cond
    ((= dl1 0)
     dats2)
    ((= dl2 0)
     dats1)
    ((and (= dl1 1) (= dl2 1))
     (2dats (1-1st-dats dats1) (1-1st-dats dats2)))
    ((and (= dl1 1) (= dl2 2))
     (3dats (1-1st-dats dats1)
            (2-1st-dats dats2)
            (2-2nd-dats dats2)))
    ((and (= dl1 2) (= dl2 1))
     (3dats (2-1st-dats dats1)
            (2-2nd-dats dats1)
            (1-1st-dats dats2)))
    (else
     (_sim-error 'dats-append "Tuples too long"))))

(define (dats-ref dats nat dl)
  (cond
    ((= dl 1)
     (1-1st-dats dats)) ; (= nat 1)
    ((= dl 2)
     (cond ((= nat 1) (2-1st-dats dats))
           ((= nat 2) (2-2nd-dats dats))))
```

```
    ((= dl 3)
     (cond ((= nat 1) (3-1st-dats dats))
           ((= nat 2) (3-2nd-dats dats))
           ((= nat 3) (3-3rd-dats dats))))))


;-------------------------------------------------
; Binding-time improvements:

(define (cl->d c dl)
  (lambda (x1 x2 x3) (c (d->ps x1 dl) x2 x3)))
(define (d->cl c dl)
  (lambda (x1 x2 x3) (c (ps->d x1 dl) x2 x3)))
(define (collapse c dl)
  (let ((c1 (lambda (x1 x2 x3)
              (c (d->ps x1 dl) x2 x3))))
    (lambda (x1 x2 x3)
      ((if #t c1 (generalize c1))
       (ps->d x1 dl) x2 x3))))

(define (ps->d dats dl)
  (cond
    ((= dl 0)
     (0dats))
    ((= dl 1)
     (1dats (1-1st-dats dats)))
    ((= dl 2)
     (2dats (2-1st-dats dats) (2-2nd-dats dats)))
    (else ; (= dl 3)
     (3dats (3-1st-dats dats)
            (3-2nd-dats dats)
            (3-3rd-dats dats)))))
(define (d->ps dats dl)
  ...) ; same code as body of ps->d

;-------------------------------------------------
; Auxiliary:

(define (failure f commit) (lambda () (f commit)))
(define (error? type-correct)
  (equal? type-correct 'error))
(define (loop)
  (_sim-error 'loop "Going off the deep end: diverge"))
(define (bomb) "Committed failure (bomb :-)")
```

```
;-----------------------------------------------
; Primitives and constructors:

(defprim-opaque (init! in-file out-file)
  (set! **store** (make-vector 5000 "Uninitialized"))
  (set! **last-used** -1)
  (set! **input-port** (open-input-file in-file))
  (set! **output-port** (open-output-file out-file)))
(defprim-opaque (close!)
  (close-input-port **input-port**)
  (close-output-port  **output-port**))

(defprim-opaque (allocate-cell!)
  (set! **last-used** (+ 1 **last-used**))
  **last-used**)
(defprim-opaque (update-store! location value)
  (vector-set! **store** location value))
(defprim-opaque (read-store! location)
  (vector-ref **store** location))

(defprim-opaque (input!) (read **input-port**))
(defprim-opaque (output! value)
  (write value **output-port**))

(defprim-dynamic (generalize x) x)

(defconstr (0dats)
           (1dats 1-1st-dats)
           (2dats 2-1st-dats 2-2nd-dats)
           (3dats 3-1st-dats 3-2nd-dats 3-3rd-dats))
(defconstr (init-env-cstr)
           (cons-env-cstr car-env-sel cdr-env-sel))
(defconstr (binding-env-cstr name-env-sel dat-env-sel))
```

C The Tiny Language

This appendix presents an example of an action semantic description. The LaTeX source text of the appendix is a legal input to both the compiler generator (Cantor) of the second author [22, 20, 21] and also to our new one.

C.1 Abstract Syntax

grammar:

(1) Statement = ⟦ Identifier ":=" Expression ⟧ |
⟦ "while" Expression "do" Statement ⟧ |
⟦ Statement ";" Statement ⟧ |
⟦ Declaration "begin" Statement "end" ⟧ .

(2) Declaration = ⟦ "var" Identifier ⟧ |
⟦ "const" Identifier "=" natural ⟧ |
⟦ Declaration ";" Declaration ⟧ .

(3) Expression = ⟦ "nat" natural ⟧ | Identifier |
⟦ Expression Operation Expression ⟧ .

(4) Operation = "+" | "<" .

(5) Identifier = token .

C.2 Semantic Functions

introduces: execute _ , establish _ , evaluate _ ,
operation-result _ , id _ .

C.2.1 Statements

- execute _ :: Statement → act .

(1) execute ⟦ I:Identifier ":=" E:Expression ⟧ =
evaluate E then
store it in the cell bound to id I .

(2) execute ⟦ "while" E:Expression "do" S:Statement ⟧ =
unfolding
| | evaluate E
then
| | | check it then execute S then unfold
| | or check not it .

(3) execute ⟦ S_1:Statement ";" S_2:Statement ⟧ =
execute S_1 and then execute S_2 .

(4) execute ⟦ D:Declaration "begin" S:Statement "end" ⟧ =
| furthermore establish D
hence execute S .

C.2.2 Declarations

- establish _ :: Declaration → act .

(1) establish ⟦ "var" I:Identifier ⟧ =
    allocate integer cell then bind id I to it .

(2) establish ⟦ "const" I:Identifier "=" n:natural ⟧ =
    bind id I to n .

(3) establish ⟦ D_1:Declaration ";" D_2:Declaration ⟧ =
    establish D_1 before establish D_2 .

C.2.3 Expressions

- evaluate _ :: Expression → act .

(1) evaluate ⟦ "nat" n:natural ⟧ = give n .

(2) evaluate I:Identifier =
    give the integer bound to id I or
    give the integer stored in the cell bound to id I .

(3) evaluate ⟦ E_1:Expression O:Operation E_2:Expression ⟧ =
    | evaluate E_1 and then evaluate E_2
    then give operation-result O .

C.2.4 Operations

- operation-result _ :: Operation → dependent datum .

(1) operation-result "+" =
    sum(the given integer #1, the given integer #2) .

(2) operation-result "<" =
    (the given integer #1) is less than
    (the given integer #2) .

C.2.5 Identifiers

- id _ :: Identifier → token .

(1) id k:token = k .

D Tiny Example Program

This appendix first presents an example program in the Tiny language. We use an appropriate concrete syntax.

```
const n = 10;
var x
begin
  x := 0;
  while x < n do
    x := x + 1
end
```

This program can be expanded to an action and then type checked by the action type-checker. In the following we present the resulting, annotated action. We use an appropriate concrete syntax, rather than the abstract syntax of Appendix A. For readability, we have rearranged the action using some of the algebraic laws of actions [18].

```
execute ⟦ "const" "n" "=" 10 ";" ... "end" ⟧ =
    | furthermore
    | | | bind "n" to 10 (ok)
    | | before (("n") 0 0)
    | | | allocate integer cell then bind "x" to it (ok)
    hence (("x" "n") 0 0)
    | | give 0 (ok) then
    | | store it in the cell bound to "x" (ok)
    | and then (0 0)
    | | unfolding (0)
    | | | | | give the integer bound to "x" (error) or (1)
    | | | | | give the integer stored in the cell bound to "x" (ok)
    | | | | and then (1 1)
    | | | | | give the integer bound to "n" (ok) or (1)
    | | | | | give the integer stored in the cell bound to "n" (error)
    | | | then give (the given integer #1 (2)) is less than
    | | | |           (the given integer #2 (2)) (ok)
    | | | then
    | | | | | check not it (ok)
    | | | | or (0)
    | | | | | | check it (ok)
    | | | | | then
    | | | | | | | give the integer bound to "x" (error) or (1)
    | | | | | | | give the integer stored in the cell bound to "x" (ok)
    | | | | | | and then (1 1)
    | | | | | | | give 1 (ok)
    | | | | | then give sum(the given integer #1 (2),
    | | | | | |                 the given integer #2 (2)) (ok)
    | | | | | then store it in the cell bound to "x" (ok)
    | | | | | then unfold (0)
```

This annotated action can then be compiled by the action compiler. The result is a 105 lines Scheme program (which we omit). We then partially evaluate that Scheme program (with no static input) once more. The result is the following 31 lines Scheme program (we have renamed bound variables, for readability).

```
(define (int in-file out-file)
  (define (ev-act c ten e x f)
    (let ((xval1 (read-store! x)))
      (if (equal? xval1 "Uninitialized")
        (f #t)
        (let ((g (< xval1 ten)))
          (cond ((equal? (not g) #t) (c (0dats) '() #t))
                ((equal? g #t)
                 (let ((xval2 (read-store! x)))
                   (if (equal? xval2 "Uninitialized")
                     (f #t)
                     (begin
                       (update-store! x (+ xval2 1))
                       (ev-act
                         (lambda (dats vs commit)
                                 (c (0dats) vs commit))
                         ten
                         e
                         x
                         f)))))
                (else (f #t)))))))
  (init! in-file out-file)
  (let* ((g (allocate-cell!))
         (vs (append (list g) '(10)))
         (x (list-ref vs 0)))
    (update-store! x 0)
    (ev-act
      (lambda (dats vs commit) (close!) '__completed)
      (list-ref vs 1)
      (lambda (dats commit) (close!) '__escaped)
      x
      (lambda (ct) (close!) '__failed))))
```

The function `int` is a specialized version of the function `int` in Appendix B. First, it initializes the `in-file` and the `out-file`. Then, it declares three variables `g`, `vs`, and `x`, corresponding to the `const` and `var` declarations in the Tiny program. Specifically, `g` contains a fresh cell (for the variable `x` in the Tiny program), `vs` contains a list of values (the cell and the value 10), as explained in Section 4, and `x` also contains the cell.

The partial evaluator does *not* split `vs` into two separate variables because one of its elements (the cell) is dynamic. The function `int` now proceeds with storing the value 0 in the cell (corresponding to '`x := 0`' in the Tiny program). Then it calls the function `ev-act` that corresponds to the `while` loop in the Tiny program.

The function `ev-act` is a specialized version of the function `ev-act` in Appendix B. It is called with five arguments. The first, third, and fifth are the complete, escape, and fail continuations, respectively. The second argument is the value 10, and the fourth argument is the cell for the variable `x` in the Tiny program. When `ev-act` is called recursively, only the first argument is changed, and that in a trivial way. The new continuation is '`(lambda (dats vs commit) (c (0dats) vs commit))`'. The only call of the continuation `c` is '`(c (0dats) '() #t)`' so the first argument to `c` is always `(0dats)` anyway. Clearly, the continuation '`(lambda (dats vs commit) (c (0dats) vs commit))`' could be replaced by simply `c`. The specializer does not do that because the data is passed to a dynamic continuation.

The Scheme program contains superfluous code for checking that the Tiny variable `x` is indeed initialized. A straightforward analysis of actions might annotate all uses of cells with conservative information about whether they are initialized or not. This could then be exploited by the action interpreter and lead to shorter target programs.

The Scheme program also contains superfluous code to distinguish failures: the `commit` parameter of the continuations. In the 105 lines Scheme program that we omitted, the `commit` parameters was frequently used in tests. Fortunately, the second partial evaluation got rid of those tests, making the resulting target program considerably shorter and also more efficient.

References

[1] Lars Ole Andersen. Self-applicable C program specialization. In *Proc. of PEPM'92, Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University).

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.

[3] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California. LISP Pointers V, 1*, pages 1–10, June 1992.

[4] Anders Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, April 1993. Included in Similix 5.0 distribution.

[5] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[6] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *Proc. CC'92, 4th International Conference on Compiler Construction, Paderborn, Germany*, pages 95–109. Springer-Verlag (*LNCS* 641), 1992.

[7] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Eighteenth Symposium on Principles of Programming Languages*, pages 14–24, 1991.

[8] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995. Preliminary version in Proc. PEPM'94, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 11–20, Orlando, Florida, June 1994.

[9] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1991.

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[11] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Proc. Rewriting Techniques and Applications*, pages 225–282. Springer-Verlag (*LNCS* 202), 1985.

[12] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Sym-*

posium on Principles of Programming Languages. Albuquerque, New Mexico, pages 258–268, 1992.

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

[14] Karoline Malmkjær. Towards efficient partial evaluation. In *Proc. PEPM'93, Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark*, 1993. To appear.

[15] Peter D. Mosses. SIS—semantics implementation system. Technical Report Daimi MD–30, Computer Science Department, Aarhus University, 1979. Out of print.

[16] Peter D. Mosses. Unified algebras and action semantics. In *Proc. STACS'89*, pages 17–35. Springer-Verlag (*LNCS* 349), 1989.

[17] Peter D. Mosses. An introduction to action semantics. Technical Report DAIMI PB–370, Computer Science Department, Aarhus University, 1991. Lecture Notes for the Marktoberdorf'91 Summer School, to be published in the Proceedings of the Summer School by Springer-Verlag (Series F).

[18] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992. *Number 26 Tracts in Theoretical Computer Science.*

[19] Peter D. Mosses and David A. Watt. The use of action semantics. In *Proc. IFIP TC2 Working Conference on Formal Description of Programming Concepts III (Gl. Avernæs, 1986)*, pages 135–163. North-Holland, 1987.

[20] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proc. ICCL'92, Fourth IEEE International Conference on Computer Languages*, pages 117–126, Oakland, California, April 1992.

[21] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Computer Science Department, Aarhus University, 1992.

[22] Jens Palsberg. A provably correct compiler generator. In *Proc. ESOP'92, European Symposium on Programming*, pages 418–434. Springer-Verlag (*LNCS* 582), Rennes, France, February 1992.

[23] Lawrence Paulson. A semantics-directed compiler generator. In *Ninth Symposium on Principles of Programming Languages*, pages 224–233, 1982.

[24] Mitchell Wand. A semantic prototyping system. In *Proc. ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 213–221. Sigplan Notices, 1984.

[25] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, 1993.

[26] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

[27] David A. Watt. Personal communication. 1992.