Generating a Compiler for a Lazy Language by Partial Evaluation

Jesper Jørgensen *
DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø
Denmark
e-mail: knud@diku.dk

Abstract

Compiler generation is often emphasized as being the most important application of partial evaluation. But most of the larger practical applications have, to the best of our knowledge, been outside this field. Especially, no one has generated compilers for languages other that small languages. This paper describes a large application of partial evaluation where a realistic compiler was generated for a strongly typed lazy functional language. The language, that was called BAWL, was modeled after the language in Bird and Wadler [BW88] and is a combinator language with pattern matching, guarded alternatives, local definitions and list comprehensions. The paper describes the most important techniques used, especially the binding time improvements needed in order to get small and efficient target programs. Finally, the performance of the compiler is compared with two compilers for similar languages: Miranda and LML.

Keywords

Compiler generation, partial evaluation, binding time improvements, lazy functional languages.

1 Introduction

This paper describes a large application of partial evaluation. Our aim was to try to write a prototype compiler for a strongly typed lazy functional language. We wanted to see how far one could get by using partial evaluation as a compiler generation tool and what problems this might involve. The language chosen for this purpose was called BAWL[1] and was modeled after the language in Bird and Wadler [BW88], a language in the class of languages that also includes KRC [Tur82], Orwell [Wad85], Miranda[2] [Tur86] and Haskell [HW90]. A parser was generated by using the standard UNIX[3] facility Yacc [Joh75]. No type checker has been written, but its existence is supposed such that the code generation part could assume that it was working on well typed programs. The code generating part of the compiler, which is the one described in this paper, was generated by using partial evaluation. The partial evaluator used was Similix [BD91] [Bon90], an *autoprojector* (self-applicable partial evaluator) for a higher order subset of Scheme [RC86], including side effects on global variables. This means that the target language for the compiler is Scheme. Since efficient compilers exist for Scheme we can translate our source programs into sufficiently low level code by first translating them into Scheme code and then compiling the Scheme code.

1.1 Outline of the paper

The paper is organized as follows. After a short section on notation, we introduce the notion of compiler generation. Then in section 3 we introduce the language and in section 4 we describe the interpreter and describe how the compiler was generated. In section 5 we describes the binding time improvements used to improve the performance of the compiler. In section 6 we show an example of a source program and its corresponding target code. In section 7 we illustrate the performance of the compiler and in section 8 we discuss further improvements of the compiler and partial evaluation as a compiler generation tool. Section 9 contains a conclusion.

1.2 Notation

We will use denotational semantics in the style of Schmidt [Sch86] when describing the semantics of programs and program constructs.

The notation (L P d) denotes the result of running the L-program P on data d, e. g. (Scheme P d) denotes the result of running the Scheme-program P on input d.

1.3 Prerequisites

It is assumed that the reader has a basic knowledge of functional programming, partial evaluation and compiler generation by partial evaluation, e.g. as presented in [JSS85] or [JSS89].

2 Compiler generators

A compiler generator is a program (or system) that given some machine readable formal description of a programming language produces a compiler for that language. The formal description can be of many forms: a denotational semantics, an attribute grammar, a combinator based semantics, some form of an operational semantics, etc.

*This work was supported by ESPRIT Basic Research Actions project 3124 "Semantique"

[1] This acronym for Bird And Wadler Language is due to Kristoffer H. Rose.

[2] Miranda is a trademark of Research Software Ltd.

[3] UNIX is a trademark of Bell Laboratories.

2.1 Compiler generation by partial evaluation

Partial evaluators are not compiler generators, but one of the primary examples of their uses has certainly been compiler generation. This is also supported by the fact that one of the main motivations for making partial evaluators self-applicable is that it makes compiler generation possible. Today self-applicable partial evaluators are available for many types of languages, e.g. imperative languages (a flowchart language [GJ89]) and functional languages (Mixwell [JSS85] and Scheme [Bon91b] [Con88]).

The way one generates a compiler for a given language by partial evaluation is by writing an interpreter Int for the language. From this interpreter the compiler is generated by specializing the partial evaluator Mix with respect to the interpreter:

Compiler = L Mix (Mix,Int)

One gets a compiler generator by specializing the partial evaluator with respect to itself:

Cogen = L Mix (Mix,Mix)

This means that the specification is the interpreter and the specification language is the source language of the partial evaluator. It also means that the choice of partial evaluator will fix the specification language. We have used an approach that uses a specification close to a denotational one and therefore a partial evaluator for a higher order functional language is the most suitable. For this reason and because of its in-house availability, Similix is an appropriate choice of partial evaluator.

2.2 Similix

This section describe the features of Similix that are important for understanding the ideas of this paper. As mentioned before, Similix is a self-applicable partial evaluator for a higher order subset of Scheme including side effects on global variables. When we specialize a program we have to specify some static input and a name of a function in the program, called the *goal function*. The goal function is the function that we want to specialize with respect to the static input.

Similix uses *monovariant* binding time analysis which has the effect that residual functions are only generated for one set of binding time values for the arguments.

Similix does not handle *partially static data structures*: This means that Similix does not perform simple reduction like:

`(car (cons` $expr_1$ $expr_2$`))` $\Rightarrow$ $expr_1$

The fundamental reason for this is that it is hard to handle such transformations in a semantically safe way in a strict language (however, a solution is outlined in [Bon92]). If evaluation of $expr_2$ may not terminate then this implies that the left hand side of the rule may not terminate, but the same does not hold for the right hand side (this gets even worse if $expr_2$ contains side effects).

3 The BAWL language

We will not give a formal description of the syntax of the BAWL language, but one can be found in [Jor91a] or in Appendix B. A program is called a script as in Orwell or Miranda and a script consists of definitions of types, type aliases, functions and conformals (constants defined via patterns, e.g. (x,y)=(1+2,fib 20) or `primes` in the example below; for the understanding of this paper, just think of conformals as zero arity functions). Here is a small example of a script:

```
fac 0 = 1
fac n = n*fac(n-1)
fib n = 1,                    if n<=1
      = fib(n-1)+fib(n-2), otherwise
primes = sieve [2..]
         where
           sieve (p:x) = p:sieve[n|n<-x;n mod p>0]
```

`fac` is the factorial function, `fib` a naive version of the fibonacci function and `primes` a conformal defined to be the infinite list of primes. As can be seen from the example, scripts can contain literals (constants), variables, constructors, operators, applications, list comprehensions and arithmetic sequences. The languages also supports tuples. Patterns may include literals, variables, constructor patterns and tuples. Some of the important features that the language does not include are input facilities and modules. The boolean constructors `True` and `False` and the list constructors : and `[]` are the only constructors predefined in the initial constructor environment.

4 Generating the compiler

The compiler was generated in the following way. First a denotational semantics was written for the language and by a simple transliteration of this into Scheme, a naive interpreter was obtained. This interpreter was then rewritten by a series of *binding time improvements* (see section 5) to yield the final interpreter. All of this process was done by hand. The compiler was then generated by machine from the final version of the interpreter by partial evaluation.

4.1 Semantics of the language

Usually when we assign meaning to a program we do this by specifying the function that the program computes. This is done under the assumption that the program has some sort of main function that is the meaning of the entire program. But often one also wants to run programs in an interpreted environment (for development and debugging), and in this case it seems better to assign another meaning to a program. The meaning of a BAWL program is therefore an environment mapping function names to their meaning. We can express this a little more formally by using a denotational style semantics:

$$\mathrm{S}[\![\text{script}]\!] = \\ \mathbf{fix}(\lambda\phi.[\mathrm{f}_j \rightarrow \lambda v_1 ... \lambda v_{n_j}.\mathrm{E}[\![expr_j]\!][x_{j,i} \rightarrow v_i]\rho_0\phi]\phi_0)$$

where S is the function assigning meaning to programs. f_j ranges over all functions defined in the script, n_j is the arity of f_j, ϕ_0 an initial function environment and ρ_0 an initial variable environment. The function E that assigns meanings to expressions is shown in Appendix C. We have left out the semantics of constructors, pattern matching, guards and local definitions to make the description simpler. A detailed description of these aspects can be found in [Jor91a]. Now it is simple to assign meaning to expressions in an interpreted environment:

$$\mathrm{I}[\![\text{script}]\!][\![expr]\!] = \mathrm{E}[\![expr]\!]\rho_0(\mathrm{S}[\![\text{script}]\!])$$

4.2 The structure of the interpreter

The overall structure of the interpreter follows the semantics. It takes a script and an expression `expr` as input and evaluates `expr` in the scope of the environment created by the function `S`. `E` is the evaluation function for expressions, `print` is the print routine driving the evaluation.

```
(define (I script expr)
  (print (E expr init-venv (S script))))
```

`S` takes a script and returns a function environment,

```
(define (S script)
  (fix
    (lambda (fenv)
      (lambda (fun)
        (if (defined-in-script fun script)
            (let* ([def (lookup-def fun script)]
                   [expr (lookup-expr def)]
                   [formals (lookup-formals def)])
              (make-function
               formals expr init-venv fenv))
            (init-fenv fun))))))

(define (make-function formals expr venv fenv)
  (if (null? formals)
      (E expr init-venv fenv)
      (lambda (v)
        (make-function
         (cdr formals) expr
         (upd (car formals) v venv) fenv))))
```

where the function `fix` is the standard applicative fixed point operator defined as:

```
(define (fix f) (lambda (x) ((f (fix f)) x)))
```

That `fix` actually produces the right fixed point is beyond the scope of this paper, for details see [Jor91a].

Appendix D shows the `E` function as it looks in the first version of the interpreter.

4.3 Generating the compiler

If we want to generate a compiler from the interpreter `I`, all we have to do is apply the compiler generator cogen to `I` and we are done. But there is a problem with this approach, since `expr` in `I` is dynamic (we only know the script when compiling) while `expr` in `make-function` is static. This will make the first argument to `E` dynamic since the binding time analysis of Similix is monovariant. This spoils the whole idea of compiler generation since we want `expr` in `make-function` to be static to get some specialization of expressions in the script done. We will therefore choose a slightly different approach: we will only apply cogen to a program without the function `I` and make `S` our goal function. This will give us a compiler that given a script *scr* produces a residual program with a specialized version `S-0` of `S`:

```
(define (S-0)
  (lambda (fun_0)
    ...))
```

That is, the compiled script is a Scheme program with a function `S-0` that when called returns a function environment.

It might seem strange that we want to partial evaluate `S`, when `S` has only static input. One might think that we could just apply `S` to the script instead. There are two reasons that this does not give the desired result. Firstly, applying `S` to a script will not generate a piece of code that can be saved on a file. Instead it will return an internal represention of a function. Secondly, standard evaluation of lambdas in Scheme is to weak head normal form and thus does not evaluate under lambdas. On the other hand, a partial evaluator does perform static reduction under lambdas (controlled by the partial evaluator's particular strategy), so applying `(S-0)` to a value will be more efficient than applying `(S` *scr*`)` to a value.

Now that we have removed the interpreter function `I` from the program we can define a slightly different version that will be used when running compiled scripts:

```
(define (I fenv expr)
  (print (E expr init-venv fenv)))
```

This takes a function environment `fenv` instead of a script as before. This function environment can be the result of a compilation or can be produced by the function `S`. We can now define a function `run` that takes an expression `expr` and the name of a file which holds a compiled script and evaluates the expression with respect to the script:

```
(define (run expr script-file-name)
  (begin
    (load script-file-name)
    (I (S-0) expr)))
```

The description we have given so far in this section is somewhat simplified, since `S` also returns a constructor environment and some tables, but we have left out these details, since they are not important for the general ideas.

4.4 Lazy evaluation

When translating into Scheme, functions of our source language functions become Scheme functions and are therefore strict. But we want our implementation to be lazy. Often *call by need* is called *lazy evaluation*, since it postpones evaluation of arguments until these become needed. But we want more than call by need, that is, we want our implementation to be lazy in more than evaluation of arguments. Assume that we write the following version of the fibonacci function:

```
fib n = fl!n
        where
          fl = 1:1:[fl!n + fl!(n+1)|n<-[0..]]
```

Here `fl` is not an argument but a conformal, and we surely do not want to calculate the conformal `fl` every time it is used.

To summarize: we want our implementation to be lazy in two ways, evaluation of arguments and evaluation of conformals.

4.5 Achieving call by need

Let us first see how we can achieve call by need. Call by need can be viewed as an optimization of call by name and call by name can easily be simulated in Scheme. This is done by suspending the evaluation of an argument expression *expr* by putting a lambda abstraction around it:

```
(lambda () expr)
```

and we say that we have suspended the evaluation of *expr*. Then, when the value of the argument is needed, instead of just using the argument a we have to apply it first:

```
(a)
```

That is, we force the evaluation of the expression in a. In this way an argument is evaluated *each* time it is needed. There are several methods to avoid this, but the one presented here (taken from [RC86])[4] is both simple and very efficient to execute[5]. For delaying an argument we now use:

```
(save (lambda () e))
```

where `save` is defined as:

```
(define (save s)
  (let ([v '()] [tag #f])
    (lambda ()
      (begin
        (unless tag
                (set! v (s))
                (set! tag #t))
        v))))
```

In this way the argument expression is only evaluated if the tag is true and the first time the argument is forced the tag is set to true. This ensures that arguments are evaluated at most once. The expression evaluation function `E` then handles application like this:

```
(cond
  ...
 ((EApply? expr)
  ((E (EApply->E1 expr) venv fenv)
   (save (lambda ()
           (E (EApply->E2 expr) venv fenv)))))
  ...)
```

where `venv` is the variable environment and `fenv` is the function environment.

4.6 Making evaluation of conformals lazy

To handle lazy evaluation of conformals we had to use a slightly different approach than used to achieve call by need. The `save` operation of section 4.5 introduces a new memory cell (the `v` and `tag` variables) where the result of evaluating the argument is going to be saved the first time it is evaluated. This works because the saved evaluation is unique, i.e. it is the argument to a specific application. But evaluation of a conformal can be initiated many places in a program (reexamine the example in section 4.4) so in order to get evaluation of the conformal shared, we have to save its value in a global memory cell. This is done in the following way: Before doing the fix point iteration in `S` we create new memory cells for all the conformals in the script (top level only) and the evaluation of the conformals are then saved in these using a slightly different version of `save` called `save-at`:

```
(define (save-at a s)
  (lambda ()
    (unless (cdr a)
            (set-car! a (s))
            (set-cdr! a #t))
    (car a)))
```

`save-at` saves the result of evaluating `s` in the cell pointed to by `a`.

5 Binding time improvements of the interpreter

A common experience among people working with partial evaluation is that not all programs are equally well suited for partial evaluation and that a certain amount of rewriting of programs are necessary to make them "partial evaluate well". This means that much of what is presented in this section is in no way particular to partial evaluation of interpreters, but rather to the method of partial evaluation used. The rewritings are semantics preserving transformations that aim to make more parts of a program static such that the partial evaluator can do more work, hence the name "binding time improvements". For further discussions of binding time improvements see [HH91], [HG91] or [Bon91b]. Here is a list of the important binding time improvements we have used:

- Using static information about dynamic data.
- Transformations into continuation passing style.
- Simulating partially static data structures to achieve *variable splitting* [Ses85].
- Removal of bindings from environments.

The last point is not really a binding time improvement, but it does improve the size and performance of the target programs. We will now explain these improvements in detail.

5.1 Using static information about dynamic data

A key point in the rewriting process is to identify static information that is not visible to the specializer and make this visible. Let us for instance assume that we have some dynamic variable `x` somewhere in a program. Since `x` is dynamic the specializer can not do any computation depending on `x`. Now it might be possible from the context in which `x` occurs to detect some information about the value of `x` depending only on static values in the program and then rewrite the program such that the specializer can utilize this information. This idea was one of the main ones introduced in the KMP example by Consel and Danvy [CD89].

Let us look at an example that illustrates this idea. The example shows a rewriting that is classic (it was already used in the original MIX project [JSS85]) and it is absolutely essential to be able to get a reasonable result. If we look at our definition of `S` of section 4.2 we definitely want `fun` to be static, since if it is not, then `def`, `expr` and `formals` all becomes dynamic and we can hardly expect the specializer to do any serious work under these conditions. But `fun` is not static, because it can clearly be bound to function names from the dynamic expression of the `I` function. The observation that saves the day is that we know which values that `fun` can possibly be bound to, namely

[4] Compiling lazy languages by partial evaluation seems to have been done for the first time in [Bon91a].

[5] I have compared several different versions that people at DIKU have come up with and this one performed best

the function names defined in `script`. If `fun` is bound to something else it is either a name defined in the initial function environment or an error. So if we rewrite `S` into

```
(define (S script)
  (let ([funs (collect-function-names script)]
        [fenv (fix ...)])
    (lambda (fun)
      (S1 fun funs fenv))))

(define (S1 fun funs fenv)
  (if (null? funs)
      (init-fenv fun)
      (let ([fun1 (car funs)])
        (if (equal? fun fun1)
            (fenv fun1)
            (S1 fun (cdr funs) fenv)))))
```

where `(fix ...)` is the original body of `S` (cf. section 4.2), then the generated function environment `fenv` is now only applied to the static `fun1` in `S1` rather than the dynamic `fun`.

The idea is used extensively in rewriting the interpreter and is essential to make pattern matching compile into efficient code. This is described in detail in [Jor91b].

5.2 Continuation passing style

The most important rewriting is transformation into continuation passing style (in some cases just tail recursive form). The importance of this has also been shown in [Dan91]. The cases that can often successfully be handled by this kind of transformation are those where static values gets caught in a dynamic context. A typical example can be a function returning several results (packed together in e.g. a list) of which some are static and some dynamic. The result will then be dynamic and the static values will then be inaccessible to the specializer. By a transformation into continuation passing style one can pass the arguments separately to a continuation which can now use the static ones of these.

Another situation in which transformation into continuation passing style may improve binding times is when functions that represent specialization points (i.e. become residual functions) return higher order values. Let us look at an example (a trivial one, but it serves the purpose of demonstrating the idea). Consider an application

```
((f x) 42)
```

where `f` is defined by:

```
(define (f x)
  (if < some test involving x >
      (lambda (z) (sub1 z))
      (lambda (z) (add1 z))))
```

Assume that `x` is dynamic, then calls to `f` will be specialization points (with Similix's current strategy) and Similix will therefore make `(f x)` dynamic. Hence the application of `(f x)` to `42` will not be performed. Let us rewrite the application into:

```
(fc x (lambda (c) (c 42)))
```

and the definition of `f` into:

```
(define (fc x c)
  (if < some test involving x >
      (c (lambda (z) (sub1 z)))
      (c (lambda (z) (add1 z)))))
```

Then application of the lambda abstractions to `42` will be performed, since the continuation is applied directly to the closure values that the original function `f` would have returned.

5.3 Partially static data structures

As stated in section 2.2 Similix does not handle partially static data structures, but in many cases these can be simulated by using higher order functions. The simplest example is partially static lists which can be simulated by using the following definitions (this special version is due to Torben Mogensen) instead of `()`, `cons` and `list-ref`:

```
(define (snil) (lambda (n c) (n)))
(define (scons a d) (lambda (n c) (c a d)))
(define (slist-ref l index)
  (l (lambda () 'error)
     (lambda (a d)
       (if (= 0 index)
           a
           (slist-ref d (sub1 index))))))
```

The effect gained by this is normally called variable splitting or *arity raising* [Rom88]. The reason that this method works is that Similix treat higher order functions, called closures, as a kind of partially static structures in that they can contain both static and dynamic subparts. When a closure is applied to its arguments during specialization, the body of the closure is specialized and the subparts may emerge again. Partial evaluators that directly handle partially static data structures will of course not need this transformation. Another way to do this improvement is to replace the data structures by environments. This idea is explained in detail in [Jor91b].

5.4 Removal of bindings from environments

This transformation is very dependent on the fact that we are working with an interpreter and will probably be very hard to automate. Assume that we have a definition of the form

```
f x y = ... (fac x) ...
        where
          fac 0 = 1
          fac n = n*fac(n-1)
```

in a source program. Then a compiler generated by Similix from a simple interpreter gives a result of form:

```
(define (fac-0 x_0 y_0)
  (lambda (n_0)
    (...
        (fac-0 x_0 y_0))))
```

The presense of the variables `x_0` and `y_0` reflect the fact that the values of `x` and `y` are in fact accessible inside the definition of `fac`. The specializer is "unaware" of the fact that the definition of `fac` does not use these values and therefore "thinks" that a call to `fac-0` can depend on these. This problem is also mentioned by Bondorf in [Bon90], but no solution was given there.

A way to solve the problem is to do a live variable analysis of `fac`'s definition. This is in fact very simple since what it boils down to is just finding free variables. The result of this analysis is then used to reduce the set of variables that are bound in environments to the set of live variables. This reduction then has to take place before evaluating `fac`'s right hand side. This method has been used with

good results to reduce the size of the target code produced by the compiler.

6 An example

In this section we give a small example of what a target function generated by the compiler may look like. The target code is shown exactly as produced by the compiler. The example is the factorial function `fac` from section 3 and the compiled version of this function looks like:

```
(define (make-fun-cl-0-3 sc_0)
  (lambda (v_1)
    (if (struct-equal? 0 (v_1))
        1
        (* (v_1)
           ((make-fun-cl-0-3 sc_0)
            (save (lambda () (- (v_1) 1))))))))
```

where `v_1` is a delayed value and the code `(v_1)` corresponds to forcing `v_1`. The code is of course not optimal, but we would need to do strictness analysis in order to improve further on the result. This will be discussed in more detail in section 8. Appendix A shows the entire target program form which the function above is taken.

7 Performance

We will illustrate the performance of the compiler in two ways. First we will show the speedup gained by partial evaluation, that is, by compilation. Second we will compare our compiler with two similar products: Miranda version 2 from Research Software Ltd. and the LML compiler of Chalmers University.

7.1 Measuring run times

All the tests were run on Sparc station 1/Sun OS 4.1. The Scheme system used was Chez Scheme Version 3.2 and the run times in Scheme were measured using the `time` function. The Miranda system used was Miranda version 2 system from Research Software Ltd. and the run times were measured using the `count` option. The version of LML was .99.3 and the run times were measured using the S option. None of the run times include garbage collection.

7.2 Speedup

We will not measure the speedup gained by partial evaluation in the usual way as the ratio between the run time of the original program (in our case the interpreter) and the run time of the residual program (in our case the target program), since this way of measuring is not entirely fair in our case. This is because our interpreter is biased towards partial evaluation 5, which means that it may run several times slower than one that is not. Instead we define the speedup as the ratio between the run time of the first interpreter (before the binding time improvements) running our source program and the run time of the target program produced by our compiler. The problems involved in performance measurements will be described in more detail in [Jor91a].

Figure 1 shows the run times and the speedups for two small examples. `fac` and `primes` have already been defined and `take` is a predefined function which takes the first `n` elements of a list. T_{Int} is the time that the interpreter used to run the program, while T_{Target} is the time it took the target program to do the same task.

Expression	T_{Int}/s.	T_{Target}/s.	Speedup
`take 100 primes`	46.3	0.67	69.1
`fac 100`	0.82	0.015	54.7

Figure 1: Run-times and speedups

7.3 Comparing BAWL with Miranda and LML

We have compared BAWL to Miranda and LML on a number of examples all taken from the Miranda manual, though some have been modified slightly to stay syntactically within LML and our language. A few results from this test are shown figure 2 and 3. A complete description of the test can be found in [Jor91a].

Expression	Run times/sec		
	Miranda	BAWL	LML
`fac 100`	0.048	0.015	.040
`fib 20`	4.43	0.96	.20
`hd (drop 99 primes)`	0.87	0.61	.11

Figure 2: Compared run times

Expression	Storage/kbytes		
	Miranda	BAWL	LML
`fac 100`	40.3	9.97	11.3
`fib 20`	2050	1230	449
`hd (drop 99 primes)`	379	410	102

Figure 3: Compared storage usage

We can in general state that our implementation is as fast or a little faster than the Miranda system and uses about the same amount of storage. We can on the other hand not compete with the graph reduction based LML system, neither with respect to speed nor use of storage.

7.4 The compiler

The compiler has 220 functions and its size is around 23.0K cells (the number of "cons" cells needed to represent it in memory) which is about twice as big as the compiler generator of Similix. It is hard to get a fair measure of the performance of the compiler itself since it performs no type checking. To give an idea, the compile time of the example of section 3 was .32 seconds. We have tried to compile larger examples (a pattern matching compiler and a binding time analysis) with good results.

8 Discussion

8.1 Further improvements of the compiler

There is still room for improvements of the compiler, both by further rewriting of the interpreter or by transformations of either the source programs or the target programs.

As described in section 4.6 all target functions get extra parameters to allow them to access the values of conformals, and this happens even if the functions never actually need to access any of these values. There are also other reasons for extra parameters that we will not mention here, but a simple liveness analysis of the source programs might help to eliminate some of these superfluous parameters.

In the Scheme code generated by the compiler many functions are curried, because these in a way inherit their

structure from the source language. Uncurrying some of these functions could improve performance considerably because tupled application is much faster in Scheme (fewer closures have to be built). Which functions to uncurry could be decided by a simple analysis of the source program and the result of this could be used by the interpreter, and this would then make these target functions appear in uncurried form.

Strictness analysis of the source program could also be used to improve the target programs. Using strictness information would in the case of the factorial function give a speedup of the generated code of about 3 times. It seems that strictness optimizations can be combined with what corresponds to a local evaluation order analysis. This can be done by keeping track of which variables have been forced and which have not. Then instead of forcing a delayed value every time it is used we should only force it the first time, then record the forced value in the environment and then subsequently use the value recorded in the environment. Using this optimization on the example of section 6 we can obtain,

```
(define (make-fun-cl-0-3)
  (lambda (v_1)
    (let ([fv_1 (v_1)])
      (if (equal? 0 fv_1)
          1
          (* fv_1
             ((make-fun-cl-0-3)
              (save (lambda () (- fv_1 1)))))))))
```

even without a strictness analysis.

Experiments also show that lambda lifting the source programs can yield good results, but this is a field that needs more investigation.

If a type-checker were written, the result of the type-checking could be used to optimize the target code, e.g. replace polymorphic equality by specific instances.

Since Scheme is dynamically typed, Scheme implementations spend a lot of time doing type checking when running programs and this time is wasted if we know that a program is "well typed." Since the target programs produced by our compiler are all well typed, it should be possible to run these without the dynamic type checking and thereby improve further on their performance. One solution could be to generate code for a different strict language like Standard ML [MTH90] and this might be possible by rewriting the back end part of Similix that generates code. Another solution would be to have a Scheme compiler in which one could turn off the run time type checking.

8.2 Partial evaluation as a compiler generation tool

For a compiler generation system to be of any use it should either provide us with a faster way to obtain compilers than by traditional hand performed methods or it should give us a safer method to obtain correct compilers.

If writing a specification is just as hard as writing the actual compiler by hand then nothing is gained. It is hard to estimate the time used to write our code since the methods used were developed during the process. The program being specialized is 587 lines long and auxiliary files (called adt-files in the Similix system) containing functions operating on the representation of value, standard environment, the interpreter, etc., contains 876 lines (in both cases without counting comments).

By the correctness of the compiler we mean with respect to the the original semantics. This correctness depends on the correctness of Similix, on the correctness of the binding time improvements and the transformation of the semantics into the interpreter. We are presently working on proving the correctness of the last two criteria.

Some of the advantages of using partial evaluation as a compiler generation tool are:

- It is generally easier to read and write an interpreter than a compiler. This makes it easier to change and maintain the compiler.
- Debugging an interpreter is easier than debugging a compiler. One can trace evaluation and one can look at both compile time data structures and run time data structures at the same time. When debugging a compiler either the compiler or the target program may fail to work; there is no such separation when debugging an interpreter. We say that there is no separation of binding time.
- You get an interpreter thrown in to boot, that is, we get both an interpreter and a compiler at the expense of writing an interpreter only. This can be useful for many purposes, especially one might instrument the interpreter with operations tracing the execution or doing statistics. This will the result in a compiler doing these operations too.
- Code generation is handled by the partial evaluator, i.e. one does not have to think about things like label or variable name generation, backpatching etc.
- Target and specification language is the same (in our case Scheme). One less language to think about makes life easier and one may assume that specification language is well known to the user of partial evaluation.

Some of the disadvantages are:

- One has less control of the code generation since this is done by the partial evaluator. This means that one can only generate code that the partial evaluator is able to produce. Also one may not be able to obtain the sharing of target code that may be possible by conventional compilers.
- Target and specification languages are the same. This is also a disadvantage since it prevent us from choosing different target languages.

In general one can say that we get the advantages at the price of some freedom in the way we can generate target code.

9 Conclusion and future work

We have shown that it is possible by partial evaluation to generate compilers for languages of a realistic size and that the compilers generated can in fact produce reasonably fast target code even when compared to handwritten compilers.

It remains to be shown whether it is possible to automate some or all of the binding time improvements discussed in section 5. Binding time improvement is certainly an interesting field and presently some research is under way in this field. We are presently looking into the problems of formalizing and proving correct the translation from denotational semantics into Scheme interpreters.

Acknowledgements

Many people have contributed in various ways; but I would especially like to thank Anders Bondorf, Carsten Gomard, John Hannan, Fritz Henglein, Neil D. Jones, John Launchbury, Lars Mathiesen, Torben Mogensen, Kristoffer Høgsbro Rose and Peter Sestoft. I would also like to thank the members of the TOPPS group at DIKU.

References

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16, 1991. To appear.

[Bon90] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In Neil D. Jones, editor, *ESOP'90, Copenhagen, Denmark. Lecture Notes in Computer Science 432*, pages 70–87, Springer-Verlag, May 1990.

[Bon91a] Anders Bondorf. Compiling laziness by partial evaluation. In [JHH91], pages 9–22, 1991.

[Bon91b] Anders Bondorf. Similix manual, system version 4.0. Included in Similix distribution, September 1991.

[Bon92] Anders Bondorf. Improving binding times without explicit cps-conversion. 1992. Forthcoming.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming. Computer Science*, Prentice-Hall, 1988.

[CD89] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[Con88] Charles Consel. New insights into partial evaluation: the schism experiment. In Harald Ganzinger, editor, *ESOP'88, Nancy, France. Lecture Notes in Computer Science 300*, pages 236–247, Springer-Verlag, March 1988.

[Dan91] Olivier Danvy. Semantics-directed compilation of nonlinear patterns. *Information Processing Letters*, 37(6):315–322, 1991.

[GJ89] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: a case study. In *Proceedings of the Twelfth IFIP World Computer Congress*, 1989.

[HG91] Carsten Kehler Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale University, New Haven, Connecticut. SIGPLAN Notices, volume 26, 9*, pages 223–233, ACM Press, June 1991.

[HH91] Carsten Kehler Holst and John Hughes. Towards binding-time improvement for free. In [JHH91], pages 83–100, 1991.

[HW90] Paul Hudak and Philip Wadler. *Report on the programming language Haskell.* Technical Report, Yale University and Glasgow University, April 1990.

[JHH91] Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990. Workshops in Computing*, Springer-Verlag, August 1991.

[Joh75] S. C. Johnson. *YACC: Yet another compiler compiler.* Technical Report 32, Bell laboratories, Murray Hill, New Jersey, 1975.

[Jor91a] Jesper Jørgensen. *Compiler Generation by Partial Evaluation.* Master's thesis, DIKU, University of Copenhagen, Denmark, student report, October 1991. Forthcoming.

[Jor91b] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In [JHH91], pages 177–195, 1991.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140, Springer-Verlag, 1985.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, 1990.

[RC86] Jonathan Rees and William Clinger. Revised report[3] on the algorithmic language scheme. *Sigplan Notices*, 21(12):37–79, December 1986.

[Rom88] Sergei A. Romanenko. A compiler generator produced by a self-applicable specialiser can have a surprisingly natural and understandable structure. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463, North-Holland, 1988.

[Sch86] David A. Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[Ses85] Peter Sestoft. The structure of a self-applicable partial evaluator. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark. Lecture Notes in Computer Science 217*, pages 236–256, Springer-Verlag, October 1985.

[Tur82] David Turner. Recursion equations as a programming language. In Darlington et al., editor, *Functional Programming and Its Applications.*, pages 1–28, Cambridge University Press, 1982.

[Tur86] David Turner. An overview of Miranda. *Sigplan Notices*, 21(12):158–166, December 1986.

[Wad85] Philip Wadler. *Introduction to Orwell.* Technical Report, Programming Research Group, University of Oxford, 1985.

A Example of a complete target program

This appendix contains a complete target program exactly as produced by the compiler. The source program was:

```
fac 1 = 0
fac n = n*fac(n-1)
primes = sieve [2..]
         where
           sieve (p:x) = p:sieve[n|n<-x;n mod p~=0]
```

and here is the target program:

```
(loadt (string-append **similix-library** "scheme.adt"))
(loadt "thunk.adt")
(loadt "bawl.adt")

(define (target-0 fenvi_0 cenvi_1)
  (let ([cfst_2 (make-cfst 1)])
    (vector
     (lambda (fn_3)
       (cond
        [(equal? fn_3 'fac) (make-fun-cl-0-3 cfst_2)]
        [(equal? fn_3 'primes)
         (save-at
          (conf-store-ref cfst_2 0)
          (lambda ()
            ((make-fun-cl-0-10) (save (lambda () (iterate+1 2))))))]
        [else (fenvi_0 fn_3)]))
     (lambda (cn_4) (cenvi_1 cn_4))
     '(fac)
     '(primes)
     ())))

(define (make-fun-cl-0-10)
  (lambda (v_0)
    (if (v:? (v_0))
        (let* ([d_2 (cdr (v_0))] [d_3 (car (v_0))])
          (inv: d_3
                (save (lambda ()
                        ((make-fun-cl-0-10)
                         (save (lambda () (lcg-0-15 (d_2) d_3))))))))
        (error 'fd
               "No matching equations for function: ~s"
               'sieve))))

(define (lcg-0-15 v_0 venv_1)
  (if (vnil? v_0)
      'nil
      (let ([d1_2 (car v_0)])
        (if (not-equal? (modulo (d1_2) (venv_1)) 0)
            (inv: d1_2 (save (lambda () (lcg-0-15 ((cdr v_0)) venv_1))))
            (lcg-0-15 ((cdr v_0)) venv_1)))))

(define (make-fun-cl-0-3 sc_0)
  (lambda (v_1)
    (if (struct-equal? (v_1) 0)
        1
        (* (v_1)
           ((make-fun-cl-0-3 sc_0) (save (lambda () (- (v_1) 1))))))))
```

B Syntax of BAWL

Figure 4 shows the syntax of the language BAWL that the compiler is able to handle. The description is deliberately kept in a form that should be short and readable. Some parentheses may be omitted e.g. in applications, the guard may be omitted in case there is only one alternative, and in the last alternative **if** `True` may be replaced by **otherwise**. In scripts, definitions and alternatives can separated by newlines instead of ;'s and offset rules like those of Miranda hold for the scope of `where`'s. Local definitions may not contain type definitions. There are some predefined type names like: num, string, char and bool.

scr	::=	*def*{;*def*}	(script)
def	::=	*fun eq*	(function definition)
	|	*pat* `=` *rhs*	(conformal definition)
	|	*tname* {*tvar*} `::=` *cdef*{`|` *cdef*}	(type declaration)
	|	*tname* {*tvar*} `==` *texpr*	(type alias)
cdef	::=	*con* {*atexpr*}	(constructor definition)
texpr	::=	*atexpr*	(type expression)
	|	*tname* {*atexpr*}	(type construction)
	|	*texpr* `->` *texpr*	(function type)
atexpr	::=	*tname*	(type name)
	|	*tvar*	(variable)
	|	(*texpr*$_1$,...,*texpr*$_n$)	(tuples, n=0 or n≥2)
	|	[*texpr*$_1$,...,*texpr*$_n$]	(lists, n≥0)
eq	::=	{*pat*} *rhs*	(equations)
rhs	::=	*alt*{; alt} {**where** *scr*}	(right hand side)
alt	::=	`=` *expr*, **if** *guard*	(alternatives)
	|	`=` *expr*, **otherwise**	(last alternative)
guard	::=	*expr*	(guard)
pat	::=	*lit*	(literal)
	|	*var*	(variable)
	|	(*pat*$_1$: *pat*$_2$)	(pair)
	|	[*pat*$_1$,...,*pat*$_n$]	(lists, n≥0)
	|	(*con* {*pat*})	(constructor pattern)
	|	(*pat*$_1$,...,*pat*$_n$)	(tuples, n=0 or n≥2)
expr	::=	*lit*	(literal)
	|	*var*	(variable)
	|	*fun*	(function or conformal)
	|	*con*	(constructor)
	|	(*expr*$_1$ *op* [*expr*$_2$])	(operator application
	|	(*op* [*expr*])	or sections)
	|	(*expr*$_1$: *expr*$_2$)	(pair)
	|	[*expr*$_1$,...,*expr*$_n$]	(lists, n≥0)
	|	(*expr*$_1$,...,*expr*$_n$)	(tuples, n=0 or n≥2)
	|	(*expr*$_1$ *expr*$_2$)	(application)
	|	[*expr*$_1$ [,*expr*$_2$] .. [*expr*$_2$]]	(arithmetic sequences)
	|	[*expr*`|` *qual* {;*qual*}]	(list comprehensions)
qual	::=	*expr*	(filter)
	|	*pat*`<-`*expr*	(generator)
op	::=	`+,-,*,/,div,mod,^,=,~=,>,<,<=,>=,<>,~,&,\/,#,!,++,--,.`	

Figure 4: Syntax of language

C Semantics of expressions

Semantic algebra:

Domain:

Value = (Basic + List + Tuples + Constructs + Function)$_\perp$

Basic = Integer + Float + Char + ...

List = ...

Tuples = ...

Constructs = ...

Function = Value $\rightarrow$ Value

$\rho \in$ VEnv = Variable $\rightarrow$ Value (variable environment)

$\phi \in$ FEnv = FunctionName $\rightarrow$ Value (function environment)

$\gamma \in$ CEnv = ConstructorName $\rightarrow$ Value (constructor environment)

Valuation functions:

E: Expression $\rightarrow$ VEnv $\rightarrow$ FEnv $\rightarrow$ CEnv $\rightarrow$ Value

$\mathrm{E}[\![lit]\!]\rho\phi\gamma = \mathrm{L}[\![lit]\!]$

$\mathrm{E}[\![var]\!]\rho\phi\gamma = \rho[\![var]\!]$

$\mathrm{E}[\![fun]\!]\rho\phi\gamma = \phi[\![fun]\!]$

$\mathrm{E}[\![con]\!]\rho\phi\gamma = \gamma[\![con]\!]$

$\mathrm{E}[\![\texttt{[]}]\!]\rho\phi\gamma = \mathrm{inNil}()$

$\mathrm{E}[\![expr_1 : expr_2]\!]\rho\phi\gamma = \mathrm{inPair}(\mathrm{E}[\![expr_1]\!]\rho\phi\gamma, \mathrm{E}[\![expr_2]\!]\rho\phi\gamma)$

$\mathrm{E}[\![(expr_1, ..., expr_n)]\!]\rho\phi\gamma = \mathrm{inTuple}(\mathrm{E}[\![expr_1]\!]\rho\phi\gamma, ..., \mathrm{E}[\![expr_n]\!]\rho\phi\gamma)$

$\mathrm{E}[\![expr_1\ expr_2]\!]\rho\phi\gamma = \mathrm{E}[\![expr_1]\!]\rho\phi\gamma(\mathrm{E}[\![expr_2]\!]\rho\phi\gamma)$

$\mathrm{E}[\![as]\!]\rho\phi\gamma = \mathrm{AS}[\![as]\!]\rho\phi\gamma$, where as is an arithmetic sequences

$\mathrm{E}[\![lc]\!]\rho\phi\gamma = \mathrm{LC}[\![lc]\!]\rho\phi\gamma$, where lc is a list comprehension

L: Literal $\rightarrow$ Value (omitted)

AS: Expression $\rightarrow$ VEnv $\rightarrow$ FEnv $\rightarrow$ CEnv $\rightarrow$ Value (omitted)

LC: Expression $\rightarrow$ VEnv $\rightarrow$ FEnv $\rightarrow$ CEnv $\rightarrow$ Value (omitted)

Figure 5: Semantics of expressions

D Interpretation of expressions

This appendix shows the part of the interpreter that corresponds to the valuation function defined in Appendix C.

```
(define (E expr venv fenv cenv)
  (cond
   ((ELit?   expr) (L (ELit->lit expr)))
   ((EVar?   expr) (my-force (venv (EVar->var expr))))
   ((EFun?   expr) (fenv (EFun->fun expr)))
   ((Econf?  expr) (my-force (fenv (EFun->fun expr))))
   ((ECon?  expr) (cenv (ECon->con expr)))
   ((ENil?   expr) (inVNil))
   ((E:?     expr) (inV: (EL (E:->E1 expr) venv fenv cenv) (EL (E:->E2 expr) venv fenv cenv)))
   ((ETuple? expr) (Tuple (ETuple->exp* expr) venv fenv cenv))
   ((EApply? expr) ((E (EApply->E1 expr) venv fenv cenv) (EL (EApply->E2 expr) venv fenv cenv)))
   ((EAS?    expr) (AS expr venv fenv cenv))
   ((ELC?    expr) (LC expr venv fenv cenv))
   (else           (error 'E "Syntax error in expression: ~s" expr))))

(define (EL expr venv fenv cenv)
  (save (lambda () (E expr venv fenv cenv))))

(define (my-force delayed-value) (delayed-value))
```