Generating a Pattern Matching Compiler 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

May 9, 1994

Abstract

Partial evaluation can be used for automatic generation of compilers and was first implemented in [10]. Since partial evaluation was extended to higher order functional languages [9] [2] it has become possible to write denotational semantics definitions of languages and implement these with very few changes in the language treated by partial evaluators.

In this paper we use this technique to generate a compiler for a small strict combinator language with pattern matching. First, a simple denotational specification for the language is written and a compiler is generated. This first compiler turns out not to generate too efficient code.

By changing the denotational specification, new compilers that generate more efficient code are obtained. This process can be described as generating optimizing compilers by changing specifications. The optimization concerns generation of object code for pattern matching and the final compiler does in fact generate very efficient code for this. Specifically, it treats non-uniform function definitions in a satisfactory way. The optimization performed can be viewed as being equivalent to the well-known compiler optimization called common subexpression elimination.

1 Introduction

1.1 Compiler Generation

One field in which partial evaluation have been used with considerable success is automatic compiler generation, i.e. generating compilers from interpreters [10]. Another is algorithms involving pattern or string matching [6] [5]. Since many functional languages like Haskell [8] and Miranda[1] [16] use pattern matching, it is natural to ask if it is possible, using partial evaluation, to generate a compiler that compiles function definitions with pattern matching into efficient matching code. This is the motivation for this work.

*This work was supported by ESPRIT Basic Research Actions project 3124 "Semantique"
[1]Miranda is a trademark of Research Software Ltd.

The partial evaluator used is Similix [4] [2], an autoprojector for a higher order subset of Scheme [12], including both lambda abstractions and side effects on global variables. The fact that the language is a member of the LISP family makes it well suited for directly implementing denotational semantic definitions. In this way compilers can be generated from specifications by expressing these as Scheme programs (interpreters).

The method of generating compilers from denotational specifications works, but the compilers do not always generate efficient code. Therefore one usually has to do some rewriting of the specification before generating the final compiler. This paper describes such a process for a compiler for a small strict combinator language with pattern matching.

One could say that the method is a way to construct optimizing compilers, but where all the work is done by changing the specification. This has some advantages compared to the usual methods:

- It is conceptually simpler.
- It is easier to prove correct.
- The optimizer gets integrated into the compiler (no separate optimization phase).

A key point in the rewriting process is to identify static information in the specification (interpreter) and use this to do more work at partial evaluation time. Partial evaluation in general represents not only one evaluation of a program, but many. At a given point in the process of partial evaluation there might be certain statically determinable information on dynamic variables that the specializer is not using. It is this information that should be identified and made visible to the specializer by incorporating it as static data in the interpreter.

The specification presented in this paper will be rewritten three times and the three versions of the compiler will be presented and the problems they solve will be described.

In the description of the specifications the focus is on the part concerning pattern matching. The first version is a fairly straightforward implementation with only few minor changes from the denotational definition.

The second version is obtained from the first by changing it to reuse results of tests on dynamic data, a method also used in [5]. The naïve matching algorithm of the first version tries to match one alternative (list of patterns in a function definition) in turn. No information on failed attempts is reused when matching new alternatives. A way to describe results of previously performed test is devised and is used to achieve efficient matching code in target programs.

The third version solves a problem not present in [5], i.e. the decomposing of dynamic values into substructures. Programs should only decompose some dynamic value into its subparts at most once; a way to ensure this is to introduce occurrences as pointers to dynamic structures.

Finally a way to solve problems of code duplication is discussed. The problem occurs if definitions have overlapping alternatives. In that case the same code can be present many times in the target programs.

In terms of compiler optimization all these changes can be viewed as common subexpression elimination in the code generated from the first specification.

1.2 Outline of the Paper

The paper is organized as follows. After a short section on notation, we introduce the combinator language and its denotational semantics. In section 3 we develop the three versions of the compiler, we discuss how to solve the problem of code duplication, and we give some statistics on the compiler. In Section 4 we look at possible extensions of the work, and in section 5 we give a comparison with related work. Section 6 contains a conclusion.

1.3 Prerequisites

Knowledge of partial evaluation [10] and of denotational semantics [13] is required.

1.4 Notation

In denotational definitions v* is an abbreviation of a sequence of v's. If D is the name of a syntactical domain then D* is the domain of sequences of elements from D. Lists of values are written in a Prolog like style, e.g. [1,2,3] and $[h|t]$. Otherwise the notation follows [13] where a conditional is written $\_ \rightarrow \_ [] \_$, function updating $[\_ \mapsto \_]\_$ and strict lambdas $\underline{\lambda}$v.e is a shorthand notation for λv.v=$\bot \rightarrow \bot [] e$. ρ is a variable environment and ϕ a function environment. Continuations are written κ. Some specification of domains have been left out for the sake of simplicity. Thus, in the second version the specification domains of continuations are left out because these differ from valuation function to valuation function; S-cont and F-cont stand for any continuation.

Programs are written in the subset of Scheme handled by Similix [2]. Predicates on sorts always have prefix "`is`" and suffix "`?`" and functions decomposing structures into substructures are written "Structure-sort"-"Substructor-sort", e.g. `Patcst-C` returns the constant when applied to a constant pattern. Injection functions into sorts are written with prefix "`in`". The definition of these primitive operations are left out, but it should be clear from the context what they do. upd-env and init-env are syntactic extensions:

```
(extend-syntax (init-env) ((init-env) (lambda (v) (error ...))))

(extend-syntax (upd-env)
  ((upd-env v w r) (lambda (v1) (if (equal? v v1) w (r v1)))))
```

In the output from Similix, variable and function names have been renamed by hand in order to make the code more readable.

2 The Combinator Language

2.1 Syntax of the Language

The syntax of the language is shown in figure 1. The language is a strict version of the curried named combinator language described in [3] extended with pattern matching. A program consists of a list of function definitions d*, each of which consists of a name F and a list of alternatives a*. An alternative is a list of patterns and an expression. Patterns can be constants, variables, and compound patterns using the constructor :. Repeated variables in one alternative are not permitted. Alternatives may overlap and are matched from left to right. Expressions can be constants, variables, function names, the constructor : applied to two expressions, or the application of an expression to another. The constructor : is not curried.

Abstract Syntax:

pgm ∈ Program, d ∈ Definition, e ∈ Expression, c ∈ Constant
v ∈ Variable, f ∈ Function-name, a ∈ Alternative, p ∈ Pattern

pgm	::= d*
d	::= (f a*)
a	::= (p* = e)
p	::= c | v | (p_1 : p_2)
e	::= c | v | f | (e_1 : e_2) | (e_1 e_2)

Figure 1: Syntax of the combinator language

Here is an example of a combinator program that will be used as source program to test the generated compilers:

```
(goal ((x : y)    = test x y))
(test (1  (2 : x) = test 1 x)
      (x  (y : 5) = x : y)
      (x  y       = y))
```

Some syntactic sugar is used here. A parser will insert parenthesis around compound right hand sides and in applications following the convention that functions associate to the left.

2.2 The Semantics of the Language

As mentioned, the intention is to construct the compiler from the denotational semantics of the language. Figure 2 shows the semantics of the language. Since the focus is on pattern matching, the semantic definition for expressions has been left out. The result of running a program is the value of a specified goal function (f_{goal}) applied to a value (v_{input}). The goal function can be any one-argument function in the program. a_f is the arity of the function f and should actually have been written arity$[\![f]\!]$. ϕ_{init} and ρ_{init} are initial environments.

Pgm defines the semantics of a program. ϕ is the function-environment and is the least fixed point over the recursively defined combinators.

There are several reasons for using continuation passing style [13] in the semantic functions for patterns. First, it is a simple way to handle backtracking. Second, it is useful to avoid using error tags. In this way all handling of error cases is left to the expression evaluation function E. An error can only occur when matching if no alternative matches the arguments (remember that the language is strict). Third, continuations can be used to return multiple results from functions in a nice way. This has two advantages; it makes specifications (interpreters) easier to read, and when specializing the interpreters values are not returned, but passed as arguments to continuations, which then gives *variable splitting* [14].

S-cont is the domain of success continuation and F-env the domain of failure continuation.

Semantic domains:

$\rho \in$ V-env = Variable $\rightarrow$ Value
$\phi \in$ F-env = Function-name $\rightarrow$ Value* $\rightarrow$ Value
$\kappa_s \in$ S-cont = V-env $\rightarrow$ Value
$\kappa_f \in$ F-cont = $Unit \rightarrow$ Value

Valuation functions:

Pgm: Program $\rightarrow$ Function-name $\rightarrow$ Value $\rightarrow$ Value
$\mathrm{Pgm}[\![pgm]\!][\![\mathrm{f}_{goal}]\!]\ \mathrm{v}_{input} = \mathrm{fix}\ (\lambda\phi.\mathrm{D}^*[\![pgm]\!]\phi)\ [\![\mathrm{f}_{goal}]\!]\ \mathrm{v}_{input}$

D*: Definition* $\rightarrow$ F-env $\rightarrow$ F-env
$\mathrm{D}^*[\![\ ]\!]\phi = \lambda\mathrm{f}.\bot$
$\mathrm{D}^*[\![(fa^*)d^*]\!]\phi = [[\![f]\!] \mapsto (\underline{\lambda}\mathrm{v}_1...\underline{\lambda}\mathrm{v}_{a_f}.\mathrm{A}^*[\![a^*]\!][\mathrm{v}_1,...,\mathrm{v}_n]\phi)](\mathrm{D}^*[\![d^*]\!]\phi)$

A*: Alternative* $\rightarrow$ Value* $\rightarrow$ F-env $\rightarrow$ Value
$\mathrm{A}^*[\![\ ]\!]\mathrm{v}^*\phi =$ "error"
$\mathrm{A}^*[\![(p^*\texttt{=}e)a^*]\!]\mathrm{v}^*\phi = \mathrm{P}^*[\![p^*]\!]\mathrm{v}^*(\lambda v.\bot)(\lambda\rho.\mathrm{E}[\![e]\!]\rho\phi)(\lambda().\mathrm{A}^*[\![a^*]\!]\mathrm{v}^*\phi)$

P*: Pattern* $\rightarrow$ Value* $\rightarrow$ V-env $\rightarrow$ S-cont $\rightarrow$ F-cont $\rightarrow$ Value
$\mathrm{P}^*[\![\ ]\!]\mathrm{v}^*\rho\kappa_s\kappa_f = \kappa_s\ \rho$
$\mathrm{P}^*[\![p\ p^*]\!][\mathrm{v}|\mathrm{v}^*]\rho\kappa_s\kappa_f = \mathrm{P}[\![p]\!]\mathrm{v}\rho(\lambda\rho'.\mathrm{P}^*[\![p^*]\!]\mathrm{v}^*\rho'\kappa_s\kappa_f)\kappa_f$

P: Pattern $\rightarrow$ Value $\rightarrow$ V-env $\rightarrow$ S-cont $\rightarrow$ F-cont $\rightarrow$ Value
$\mathrm{P}[\![v]\!]\mathrm{v}\rho\kappa_s\kappa_f = \kappa_s\ ([[\![v]\!] \mapsto \mathrm{v}]\rho)$
$\mathrm{P}[\![c]\!]\mathrm{v}\rho\kappa_s\kappa_f = (\mathrm{C}[\![c]\!]\texttt{=}\mathrm{v}) \rightarrow \kappa_s\ \rho\ [\!]\ \kappa_f\ ()$
$\mathrm{P}[\![(p_1\texttt{:}p_2)]\!]\mathrm{v}\rho\kappa_s\kappa_f =$ **case** v **of**
$(\mathrm{v}_1,\mathrm{v}_2) : \mathrm{P}[\![p_1]\!]\mathrm{v}_1\rho(\lambda\rho'.\mathrm{P}[\![p_2]\!]\mathrm{v}_2\rho'\kappa_s\kappa_f)\kappa_f$
$\_ : \kappa_f\ ()$

E: Expression $\rightarrow$ V-env $\rightarrow$ F-env $\rightarrow$ Value (omitted)
C: Constant $\rightarrow$ Value (omitted)

Figure 2: Semantics of combinator language with patterns

3 Generating the compiler

To get a compiler for the language we write an interpreter *Int* for the language. This is a straightforward process. From this interpreter the compiler is generated in the usual way, i.e. by specializing the specializer with respect to the interpreter:

Compiler = *Mix* Mix Int

or by using the compiler generator:

Compiler = *Cogen* Int

where Mix is the specializer. Program names written in *italics* designate the meaning of the program, while names in the normal font designates program texts.

The static input to the interpreter, which is also the input to the compiler, is a source program and a goal function. So we run the compiler like:

Target = *Compiler* Source f_{goal}

3.1 The First Implementation

Before showing the first interpreter we make a small change in the semantics. Thus we replace the list [v_1,...,v_n] by an environment binding indices to their values (in fact we really bind $a_f - i$ to v_i, because it is simpler).

Semantic domains:
$\omega \in$ Occ = Nat
$\varrho \in$ O-env = Occ $\to$ Value
$\kappa \in$ Cont = O-env $\to$ Value

Valuation functions:
D*: Definition* $\to$ F-env $\to$ F-env
$\mathrm{D}^*[\![\ ]\!]\phi = \lambda \mathrm{f}.\bot$
$\mathrm{D}^*[\![(fa^*)d^*]\!]\phi = [[\![f]\!] \mapsto \mathrm{L\ a}_f\ (\lambda\varrho.\mathrm{A}^*[\![a^*]\!]\varrho \mathrm{a}_f\phi)]\ (\mathrm{D}^*[\![d^*]\!]\phi)$

L: $Nat \to$ Cont $\to$ Value
L 0 $\kappa = \kappa\ \lambda\omega.\bot$
L $\omega\kappa = \underline{\lambda}$v.L ($\omega$-1) $(\lambda\varrho.\kappa([\omega \mapsto \mathrm{v}]\varrho))$

A*: Alternative* $\to$ O-env $\to$ Occ $\to$ F-env $\to$ Value
$\mathrm{A}^*[\![\ ]\!]\varrho\omega\phi =$ "error"
$\mathrm{A}^*[\![(p*\texttt{=}e)a^*]\!]\varrho\omega\phi = \mathrm{P}^*[\![p*]\!]\varrho\omega(\lambda v.\bot)(\lambda\rho.\mathrm{E}[\![e]\!]\rho\phi)(\lambda().\mathrm{A}^*[\![a^*]\!]\varrho\omega\phi)$

P*: Pattern* $\to$ O-env $\to$ Occ $\to$ V-env $\to$ S-cont $\to$ F-cont $\to$ Value
$\mathrm{P}^*[\![\ ]\!]\varrho\omega\rho\kappa_s\kappa_f\ = \kappa_s\ \rho$
$\mathrm{P}^*[\![p\ p*]\!]\varrho\omega\rho\kappa_s\kappa_f\ = \mathrm{P}[\![p]\!](\varrho\ \omega)\rho(\lambda\rho'.\mathrm{P}^*[\![p*]\!]\varrho(\omega\text{-}1)\rho'\kappa_s\kappa_f)\kappa_f$

Figure 3: Changes to the semantics for version 1

The reason for building up this environment has to do with the way Similix works; the idea is to achieve variable splitting for v* in the semantics. This is a

general trick where one uses a higher order environments to represent structures of dynamic values. This will be explained by a small example where the structure is a list. Consider the expression:

```
(let ((l (cons v1 (cons v2 '()))))) (cadr l))
```

If we specialize this when `v1` and `v2` are dynamic, we get the residual expression

```
(cadr (cons v1 (cons v2 '())))
```

But if we replace the list with an environment representing the list

```
(let ((r (upd-env 1 v1 (upd-env 2 v2 (init-env))))) (r 2))
```

it specializes to the residual expression `v2`.

In case of structures that are more general than lists one has to use something more general than indices to reference the values. We shall see an example of this in section 3.3. Here *occurrences* will be used to reference parts of structured values. Why we use the name O-env for the domain of these environments will also become clear then.

Figure 3 shows the changed parts of the semantics. ϱ is the argument environment and L the function building it. L also constructs the lambda abstractions of $(\underline{\lambda}\mathrm{v}_1 ... \underline{\lambda}\mathrm{v}_{a_f}.\mathrm{A}^*\ ...)$ from the first version of the semantics. Figure 5 shows a fairly straightforward implementation of the semantic. This is the only interpreter that will be shown in the paper. Only the parts relevant for the pattern matching are shown. Since Scheme is a strict language one has to be careful how one writes the fixpoint combinator `fix`.

If we generate a compiler from the first version of the interpreter and compile the example program of section 2.1 we get the result shown in figure 4. In this figure only the function of the target program corresponding to the function `test` is shown.

```
(define (test)
  (lambda (x)
    (lambda (y)
      (if (equal? 1 x)
          (if (pair? y)
              (if (equal? 2 (car y))
                  (((test) 1) (cdr y))
                  (_P1 x y x y x y x y))
              (_P1 x y x y x y x y))
          (_P1 x y x y x y x y)))))
(define (_P1 r7 val6 sc5 sc4 sc3 sc2 fc1 fc0)
  (if (pair? val6)
      (let ((val1 (car val6)))
        (if (equal? 5 (cdr val6)) (cons r7 val1) fc0))
      fc0))
```

Figure 4: Target program of the first version

There are a few peculiarities in this program that are due to Similix's way of working. The many superfluous parameters to `_P1` are generated when Similix do variable splitting when generating residual calls; Similix's postprocessor could be extended to remove these.

```
(define (_D* D* phi)
  (if (null? D*)
      (init-env)
      (let* ((D (car D*))
             (occ (D-arity D)))
        (upd-env (D-F D)
                 (L occ (lambda (ro) (_A* (D-A* D) occ ro phi)))
                 (_D* (cdr D*) phi)))))

(define (L occ c)
  (if (zero? occ)
      (sc (lambda (w) '()))
      (lambda (v)
        (L (sub1 occ) (lambda (r) (sc (upd-env occ v r)))))))

(define (_A* A* occ ro phi)
  (if (null? A*)
      (error '_A* "No matching alternatives ~s" '-)
      (let* ((A (car A*)))
        (_P* P* occ ro (init-env)
             (lambda (r) (_E (A-E A) r phi))
             (lambda () (_A* (cdr A*) occ ro phi))))))

(define (_P* p* occ ro r sc fc)
  (if (null? p*)
      (sc r)
      (_P (car p*) (ro occ) r
          (lambda (r1) (_P* (cdr p*) (sub1 occ) ro r1 sc fc))
          fc)))

(define (_P p val r sc fc)
  (cond
   ((isPvar? p) (sc (upd-env (Pvar-V p) val r)))
   ((isPcst? p) (if (equal? (Pcst-C p) val) (sc r) (fc)))
   (else ; Pattern must be :-pattern!
    (if (pair? val)
        (_P (P:-P1 p) (car val) r
            (lambda (r1) (_P (P:-P2 p) (cdr val) r1 sc fc))
            fc)
        (fc)))))
```

Figure 5: A part of version 1 of the interpreter

Even apart from this it is clear that the code is not optimal; it may perform the same test several times. If the test `(equal? 2 (car y))` fails, the first action of `_P1` is the test `(pair? val6)` which is essentially the same test as the `(pair? y)`

previously performed. The program also performs the same decomposing more than once; `(car y)` and `(car val6)` do same operation.

The problem lies in the way the semantics are written. The alternatives are matched one at a time and when matching a new alternative no results obtained by matching against earlier alternatives are reused.

3.2 The Second Implementation

The first problem that we treat is the one of redundant testing. What we need is some way to memorize what know we at a given time about the arguments we match against. For this purpose we introduce a new concept, a *description* which describes what we already know about an argument. We introduce a partial ordering on descriptions, saying that one description is less than another if the set of values it describes includes of the set of values the other describes. If we have two descriptions of some argument we can get a new description of the values by taking the least upper bound of the two descriptions. The set of values this new description describes is the intersection of the sets of values of the two original descriptions (assuming that the two descriptions do not conflict in which case it describes the empty set).

We now formalize descriptions. A description is an element in the domain Desc defined by:

$$\begin{aligned}\text{Desc} = \{\ \top, \bot\ \} &\cup \text{Constant} \cup \{\ (\mathrm{d}_1 : \mathrm{d}_1) \mid \mathrm{d}_1\ , \mathrm{d}_1 \in \text{Desc})\ \} \\ &\cup \{\ \neg S\ \mid\ S \in \wp\ (\{\ :\ \} + \text{Constant})\}\end{aligned}$$

where $\wp$ (S) indicates the powerset of S. In the following α and β are arbitrary elements in Desc. The ordering is:

$$\begin{array}{ll}
\alpha \sqsubseteq \top & \\
\bot \sqsubseteq \alpha & \\
\neg S \sqsubseteq c & \textbf{if } c \notin S \textbf{ and } c \in \text{Constant} \\
\neg S \sqsubseteq (\alpha_1{:}\alpha_2) & \textbf{if } :\notin S \\
\neg S_1 \sqsubseteq \neg S_2 & \textbf{if } S_1 \subseteq S_2 \\
(\alpha_1{:}\alpha_2) \sqsubseteq (\beta_1{:}\beta_2) & \textbf{if } \alpha_1 \sqsubseteq \beta_1 \textbf{ and } \alpha_2 \sqsubseteq \beta_2
\end{array}$$

A description can be one of: A constant c, a pair-description $(\alpha_1{:}\alpha_2)$, a nomatch description ¬S, the bottom element $\bot$ or the top element $\top$. The constant description says that the described value is the constant; the :-description says that the value is a pair with subdescriptions described by the two components of the pair. The nomatch description describes what the value is known not to match. : in this set means that the data is not a pair. There is no need for compound descriptions like (1:2) in the set of nomatch elements, because descriptions are supposed to describe results of tests. The only tests that are performed on values are `equal?` and `pair?` and therefore the only possible results of failed tests are that a value is not a given constant or that it is not a pair. $\bot$ means "no information" about the value, and $\top$ describes a conflict. The least upper bound of descriptions is given by:

$$\begin{array}{lll}
\bot \sqcup \alpha & = \alpha & \\
c \sqcup c & = c & \textbf{if } c \in \text{Constant} \\
c \sqcup \neg S & = c & \textbf{if } c \in \text{Constant } \textbf{and } \alpha \notin S
\end{array}$$

$$
\begin{array}{lll}
(\alpha_1{:}\alpha_2) \sqcup \neg S & = (\alpha_1{:}\alpha_2) & \textbf{if } : \notin S \\
\neg S_1 \sqcup \neg S_2 & = \neg(S_1 \cup S_2) & \\
(\alpha_1{:}\alpha_2) \sqcup (\beta_1{:}\beta_2) & = (\alpha_1 \sqcup \beta_1{:}\alpha_2 \sqcup \beta_2) & \textbf{if } (\alpha_1 \sqcup \beta_1) \neq \top \textbf{ and} \\
 & & \quad (\alpha_2 \sqcup \beta_2) \neq \top \\
\alpha \sqcup \beta & = \top & \textbf{otherwise}
\end{array}
$$

together with the fact that $\sqcup$ is commutative.

We now change the semantics so that whenever a test is performed, the description is updated. If the result of a test can be decided entirely from the description, the test does not have to be performed at all.

A key point is that the modifications of the description depend only on the patterns. Therefore the description becomes static. This is an example where we have identified some static information and made it visible to the specializer.

The new semantics are shown in figure 6. Now L also builds up a list of bottom descriptions having a length corresponding to the arity of the function. This list is the initial description δ^* of the argument values.

The important point in the new version is that, before trying to match the next alternative, we test if the pattern list can match values with the given description. This is done by the operation inc? which returns true if the pattern and the description are incompatible. A pattern and a description are incompatible, if the pattern can not match any value in the set of values described by the description. To see why this is a good idea, let us look at this somewhat strange function:

```
(foo (x 2 7 = 1)
     (1 2 5 = 2)
     (x y z = 5))
```

If we fail to match the `2` pattern of the first alternative we should not consider the second alternative at all, and decide this before actually starting to match the patterns of the second alternative, because there is no need to first match against the `1` pattern and then fail when matching the `2` pattern, even if this failure is statically decidable from the description. If the language were lazy instead of strict, this optimization would be incorrect because it would change the semantics (e.g. evaluation of `(foo bottom 1 1)` would terminate, even if evaluation of bottom would fail to terminate).

If we write an interpreter from this second version of the semantics and generate a new compiler, we get the result shown in figure 7 when compiling the example program of section 2.1.

We see that the problem we set out to solve, the redundant testing, has disappeared. But the decomposition problem still remains.

3.3 The Third Implementation

This section describes the idea behind the third version of the compiler, but without giving a detailed description of the semantics. The idea is to extend the argument environment also to hold values of subparts of the arguments. This new argument environment is updated with values of subparts of the values the first time these are found; they can then be retrieved whenever the subparts are used again. To

Semantic domains:

$\delta \in$ Desc

Valuation functions:

$$D^*\text{:Definition}^* \to \text{F-env} \to \text{F-env}$$
$$D^*[\![\ ]\!]\phi = \lambda f.\bot$$
$$D^*[\![\,(f a^*)\, d^*]\!]\phi = [[\![f]\!] \mapsto L\ a_f\ (\lambda\varrho.\lambda\delta^*.A^*[\![a^*]\!]\varrho\ \ a_f\delta^*\phi)](D^*[\![d^*]\!]\phi)$$

$$L{:}Nat \to \text{Cont} \to \text{Value}$$
$$L\ 0\ \kappa = \kappa\ \varrho_{init}\ [\ ]$$
$$L\ \omega\kappa = \underline{\lambda} v.L\ (\omega\text{-}1)\ (\lambda\varrho.\lambda\delta^*.\kappa\ ([\omega \mapsto v]\varrho[\bot_D|\ \delta^*])$$

$$A^*\text{:Alternative}^* \to \text{O-env} \to \text{Occ} \to \text{Desc}^* \to \text{F-env} \to \text{Value}$$
$$A^*[\![\ ]\!]\varrho\omega\delta^*\phi = \text{"error"}$$
$$A^*[\![\,(p^*{=}e)\, a^*]\!]\varrho\omega\delta^*\phi = \text{inc?}\ [\![p^*]\!][\![\delta^*]\!]\ \to A^*[\![a^*]\!]\varrho\omega\delta^*\phi$$
$$[\!]\ P^*[\![p^*]\!]\varrho\omega\delta^*(\lambda v.\bot)(\lambda\rho.E[\![e]\!]\rho\phi)(\lambda\delta^*{}_1.A^*[\![a^*]\!]\varrho\omega\delta^*{}_1\phi)$$

$$P^*\text{:Pattern}^* \to \text{O-env} \to \text{Occ} \to \text{Desc}^* \to \text{V-env} \to \text{S-cont} \to \text{F-cont} \to \text{Value}$$
$$P^*[\![\ ]\!]\varrho\omega\delta^*\rho\kappa_s\kappa_f = \kappa_s\ \rho$$
$$P^*[\![p\ p^*]\!]\varrho\omega[\delta|\delta^*]\rho\kappa_s\kappa_f =$$
$$P[\![p]\!](\varrho\ \omega)\delta\rho(\lambda\rho_1.\lambda\delta_1.P^*[\![p^*]\!]\varrho(\omega\text{-}1)\delta^*\rho_1\kappa_s(\lambda\delta^*{}_1.\kappa_f\ [\delta_1|\delta^*{}_1]))(\lambda\delta_1.\kappa_f\ [\delta_1|\delta^*])$$

$$P\text{:Pattern} \to \text{Value} \to \text{Desc} \to \text{V-env} \to \text{S-cont} \to \text{F-cont} \to \text{Value}$$
$$P[\![v]\!]v\delta\rho\kappa_s\kappa_f = \kappa_s\ ([[\![v]\!] \mapsto v]\rho)\ \delta$$
$$P[\![c]\!]v\delta\rho\kappa_s\kappa_f = \text{isC?}(\delta) \to \kappa_s\ \rho\ \delta$$
$$[\!]\ (C[\![c]\!]{=}v) \to \kappa_s\ \rho\ \text{inD}(C[\![c]\!])\ [\!]\ \kappa_f\ (\delta\sqcup \text{inD}(\neg\{C[\![c]\!]\}))$$
$$P[\![p]\!]v\delta\rho\kappa_s\kappa_f = \text{isD:?}(\delta)\ \to P{:}[\![p]\!]v\delta\rho\kappa_s\kappa_f$$
$$[\!]\ \text{pair?}(v)\ \to P{:}[\![p]\!]v\ \text{inD}(\bot_D{:}\bot_D)\rho\kappa_s\kappa_f\ [\!]\ \kappa_f\ (\delta\sqcup \text{inD}(\neg\{:\}))$$

$$P{:}\text{:Pattern} \to \text{Value} \to \text{Desc} \to \text{V-env} \to \text{S-cont} \to \text{F-cont} \to \text{Value}$$
$$P{:}[\![\,(p_1{:}p_2)\,]\!](v_1,v_2)(\delta_1{:}\delta_2)\rho\kappa_s\kappa_f =$$
$$P[\![p_1]\!]v_1\delta_1\rho$$
$$(\lambda\rho_1.\lambda\delta'_1.P[\![p_2]\!]v_2\delta_2\rho(\lambda\rho_2.\lambda\delta'_2.\kappa_s\,\rho_2\,\text{inD}(\delta'_1{:}\delta'_2))(\lambda\delta'_2.\kappa_f\,\text{inD}(\delta'_1{:}\delta'_2)))$$
$$(\lambda\delta'_1.\kappa_f\,\text{inD}(\delta'_1{:}\delta_2))$$

Figure 6: Changes to the semantics for version 2

point out the subparts, we define a concept called *occurrences*. An occurrence is a sequence of a's and d's pointing out a given subpart of some argument in such a way that the value of the subpart could be found by applying the corresponding sequence of `car`'s and `cdr`'s to the value. In order to know which argument the subpart belongs to the last element in the sequence will be the argument number. So an example of an occurrence could be [d a 2]. The environment that binds these occurrences to their values is called the *occurrence environment*.

Figure 8 shows a small part of the semantics for the last version. The central point in this version is that the decomposing of an argument into a given subpart has been moved to the point where the value of the subpart is first needed. When we need to know the value of some proper subpart of an argument, we first look at

```
(define (test)
  (lambda (x)
    (lambda (y)
      (cond
       ((equal? 1 x)
        (if (pair? y)
            (if (equal? 2 (car y))
                (((test) 1) (cdr y))
                (let ((val1 (car y)))
                  (if (equal? 5 (cdr y)) (cons x val1) y)))
            y))
       ((pair? y)
        (let ((val2 (car y)))
          (if (equal? 5 (cdr y)) (cons x val2) y)))
       (else y))))))
```

Figure 7: Target program of the second version

the description of this to see if this is $\perp$. If this is the case it means that the value has not been found before, so we have to force the computation of the subpart. For example, if the occurrence is [d a 2] the value is found by looking up the value of [a 2] in the occurrence environment and finding the right subpart of this value. The function FS is the function that finds the values of subparts.

Notice that the variable environment now binds variables to occurences. This means that updating a variable does not force any decomposing. The variable environment used in the evaluation of the right hand side of the alternative is not created until the match succeeds.

Similix's unfold strategy for let-expressions ensures that dynamic let-expressions only get unfolded when the let-variable is used exactly once. Therefore if we know that the specification (interpreter) only decomposes an argument into a given subpart at the most once and only when needed, then the same will be the case for the target programs.

If we write an interpreter from this third version of the semantics and generate a new compiler, we get the result shown in figure 9 when compiling the example program of section 2.1.

This code is now nearly optimal and looks much like the program one would write by hand. But one problem still remains to be solved.

This problem concerns the size of the code and is due to the fact that the two occurrences of `cons` both come from specialization of the `_E` function in the interpreter with respect to the expression of the second alternative in the source program. If this expression was not `x:y`, but some huge expression, we would get two, and in other cases more, huge copies of the specialized code.

The reason why we get these identical pieces of code in the residual code is that they are the result of specializing the same code with respect to different values of some static arguments where the values had no influence on the result.

The static values in question are the values of the description and the occurrence environment. The problem is in fact that we have now collected static information (e.g. the description) that we might not need, but which still influences the special-

Semantic domains:

$\rho \in$ VO-env = Variable $\to$ Occ

$\kappa_s \in$ S-cont = VO-env $\to$ O-env $\to$ Desc $\to$ Value

$\kappa_f \in$ F-cont = O-env $\to$ Desc $\to$ Value

Valuation functions:

$$\begin{aligned}
&\text{P:Pattern} \to \text{O-env} \to \text{Occ} \to \text{Desc} \to \text{VO-env} \to \text{S-cont} \to \text{F-cont} \to \text{Value}\\
&\text{P}[\![v]\!]\varrho\omega\delta\rho\kappa_s\kappa_f = \kappa_s\,([[\![v]\!] \mapsto \omega]\rho)\,\varrho\,\delta\\
&\text{P}[\![c]\!]\varrho\omega\delta\rho\kappa_s\kappa_f = \text{isC?}(\delta) \to \kappa_s\,\rho\,\varrho\,\delta\\
&\qquad\qquad \square(\delta{=}\bot_D) \to \textbf{let } \text{v=FS } \varrho\,\omega \textbf{ in}\\
&\qquad\qquad\qquad\qquad (\text{C}[\![c]\!]{=}\text{v}) \to \kappa_s\,\rho\,([\omega \mapsto \text{v}]\varrho)\,\text{inD}(\text{C}[\![c]\!])\\
&\qquad\qquad\qquad\qquad \square\kappa_f\,([\omega \mapsto \text{v}]\varrho)\,(\delta \sqcup \text{inD}(\neg\{\text{C}[\![c]\!]\}))\\
&\qquad\qquad \square\textbf{let } \text{v=}\varrho\,\omega \textbf{ in}\\
&\qquad\qquad\quad (\text{C}[\![c]\!]{=}\text{v}) \to \kappa_s\,\rho\,\varrho\,\text{inD}(\text{C}[\![c]\!])\ \square\kappa_f\,\varrho\,(\delta \sqcup \text{inD}(\neg\{\text{C}[\![c]\!]\}))\\
&\text{P}[\![p]\!]\varrho\omega\delta\rho\kappa_s\kappa_f = \text{isD:?}(\delta) \to \text{P:}[\![p]\!]\varrho\omega\delta\rho\kappa_s\kappa_f\\
&\qquad\qquad \square(\delta{=}\bot_D) \to \textbf{ let } \text{v=FS}\,\varrho\,\omega \textbf{ in}\\
&\qquad\qquad\qquad\qquad \square\text{pair?(v)} \to \text{P:}[\![p]\!]([\omega \mapsto \text{v}]\varrho)\omega\,\text{inD}(\bot_D{:}\bot_D)\rho\kappa_s\kappa_f\\
&\qquad\qquad\qquad\qquad \square\kappa_f\,([\omega \mapsto \text{v}]\varrho)\,(\delta \sqcup \text{inD}(\neg\{:\}))\\
&\qquad\qquad \square\textbf{let } \text{v=}\varrho\,\omega \textbf{ in}\\
&\qquad\qquad\quad \text{pair?(v)} \to \text{P:}[\![p]\!]\varrho\omega\,\text{inD}(\bot_D{:}\bot_D)\rho\kappa_s\kappa_f\ \square\kappa_f\,\varrho\,(\delta \sqcup \text{inD}(\neg\{:\}))
\end{aligned}$$

Figure 8: P function in the semantics for version 3

```
(define (test)
  (lambda (x)
    (lambda (y)
      (cond
       ((equal? 1 x)
        (if (pair? y)
            (let ((val1 (car y)))
              (cond
               ((equal? 2 val1) (((test) 1) (cdr y)))
               ((equal? 5 (cdr y)) (cons x val1))
               (else y)))
            y))
       ((pair? y)
        (if (equal? 5 (cdr y)) (cons x (car y)) y))
       (else y)))))
```

Figure 9: Target program of the third version

ization. One could solve this by removing superfluous parts of the description. This could be determined statically from the remaining patterns, but there is a simpler solution which is described in the next section.

3.4 Solving the Problem of Code Duplication

The code duplication problem can be solved in a simple way that gives an acceptable result where the code generated from the right-hand sides are shared. We replace the call to `_E` in `_A*` by a call to a new function `rhs` defined as follows:

```
(define (rhs e r phi)
  (_E e r phi))
```

We then force the specializer to make this function residual, i.e. not to unfold calls to it. Now the duplicated code becomes calls to specialized versions of `rhs`, and since the static arguments to `rhs` have the same values the code gets shared. Using this method we get the target program in figure 10.

```
(define (test)
  (lambda (x)
    (lambda (y)
      (cond
       ((equal? 1 x)
        (if (pair? y)
            (let ((val1 (car y)))
              (cond
               ((equal? 2 val1) (((test) 1) (cdr y)))
               ((equal? 5 (cdr y)) (rhs1 x val1))
               (else y)))
            y))
       ((pair? y)
        (if (equal? 5 (cdr y)) (rhs1 x (car y)) y))
       (else y)))))
(define (rhs1 r1 r0) (cons r1 r0))
```

Figure 10: Target program with sharing of right hand sides

Now the code generated from the right-hand sides is shared, but some of the matching code, is still duplicated (e.g. `(if (equal? 5 (cdr y)) ...)`)

3.5 The Compiler

The structure of the part of the compiler involving the pattern matching is rather complex and will not be shown. The compiler has 43 functions and the size is 49552 bytes. The compiler was run on a Sun 3/160 using Chez Scheme Version 2.0.3, and the time used to compile the example from section 2.1 was 0.43 CPU seconds, not including garbage collection.

Running the first version of the interpreter 1000 times on the example of section 2.1 with input `'(1 . (2 . (2 . (2 . (2 . 3))))` takes 18.1 sec, while running the target program on the same input 1000 times takes 0.73 sec. This gives a speedup of around 25 times.

4 Extensions

This section discusses some possible ways to extend the work presented.

4.1 Other Kinds of Patterns

Extending the pattern matching to handle other kinds of patterns poses no great problems. Repeated variables can be treated by keeping a list of variables encountered so far. When meeting a new variable one can easily decide if this is already bound and, if so, check the value. Wildcard variables can be treated as normal variables, except that no updating should take place. User defined constructors can also be handled, but is related more closely to typed languages. One needs some information about the constructors, at least the arity, to be able to generate good code.

4.2 Use of Type Information

The combinator language considered in this paper is untyped, but most of the languages it resembles are typed. One could consider using type information to get even better matching code. For example, the function append:

```
(append (()       ys = ys)
        ((x : xs) ys = x : (append xs ys)))
```

compiles to:

```
(define (append)
  (lambda (xs)
    (lambda (ys)
      (cond
       ((equal? () xs) ys)
       ((pair? xs) (cons (car xs) (((append) (cdr xs)) ys)))
       (else (error ...))))))))
```

But if the language were typed and there were type information making : and () the only two members of a family of constructors, then knowing that a program is well-typed and that some value is not equal to `()` would imply that the value must be of the form `(v1,v2)`. In this case the matching algorithm should not update the description to be $\neg\ \{()\}$, but $(\bot_D : \bot_D)$. Using this type information append now compiles to the "usual" Scheme definition:

```
(define (append)
  (lambda (xs)
    (lambda (ys)
      (if (equal? () xs)
          ys
          (cons (car xs) (((append) (cdr xs)) ys)))))))
```

A similar solution can be found to handle type declarations with more than two constructors. The final result for the append function was in fact produced by a

compiler generated from an extended version of the third interpreter.

4.3 Lazy Pattern Matching

It has been shown [3] that it is possible to use partial evaluation to generate compilers for lazy languages and it is also fairly easy to change the interpreter for the combinator language to become lazy. This amounts to inserting "force" and "delay" operations at strategic places in the interpreter and, as mentioned above (section 3.2), to leave out the test inc? that was introduced in the second version. But leaving out the test inc? means that we also have to introduce a new test to check whether a constant pattern matches a constant description.

5 Related Work

To our knowledge, the first published work on matching and partial evaluation is Emanuelson [6]. He used partial evaluation to optimize pattern matching in text manipulation. The matching involved only one pattern. Since the partial evaluator he used was not self-applicable, no compiler was generated. Also, since he did no rewriting of the interpreter, his target programs have the same problems as those of our first version.

A work similar to ours is [5] where the matching problem is to find a given substring within another string. A general string pattern matcher is rewritten several times finaly leading to the Knuth, Morris and Pratt algorithm [11]. The difference between their work and ours is that while we treat many patterns and that these are trees with variables, they only treat one one constant pattern (the substring).

The first published description of a pattern matching compiler is Augustsson's description of the techniques used in the LML compiler [1]. A good description where Augustsson's work is pressented and extended is given by Wadler in [17]. The following example is taken from this, except that the right hand sides have been slightly changed:

```
(demo (f '()      ys       = ys)
      (f xs       '()      = xs)
      (f (x : xs) (y : ys) = (f x y) : (demo xs ys)))
```

Compiling this program with the final version of the compiler extended with type information on : and () gives the following target program:

```
(define (demo)
  (lambda (f)
    (lambda (xs)
      (lambda (ys)
        (cond
         ((equal? () xs) ys)
         ((equal? () ys) xs)
         (else
          (cons ((f (car xs)) (car ys))
                ((((demo) f) (cdr xs)) (cdr ys)))))))))
```

This result is actually better than the one Wadler gets. Wadler defines a restricted class of function definitions, that he calls *uniform definitions*. A definition is uniform if the patterns are such that the order of the alternatives does not matter (regardless of the right hand sides). The compilers of Augustsson or Wadler do not in general give good matching code for non uniform definitions, i.e the resulting code may examine some arguments more than once.

6 Conclusion

It has been outlined how optimizing compilers can be generated by changing specifications. The method is demonstrated on a combinator language using pattern matching where the development of a compiler for the language was described. This compiler generates very efficient matching code and the way in which the compiler was developed ensured that the target programs possesed certain properties. These are:

- An argument to a function is only tested at most once for a given property and only when needed.
- A compound argument to a function is only decomposed at most once, and only when needed.
- No code duplication of right hand sides.
- Could be extended to use type information.

These properties ensure that the compiler is capable of treating non uniform definitions in a satisfactory way.

It is an interesting open problem whether the methods demonstrated in this paper can be made automatic. It should then be a part of the preprocess phase of the partial evaluater, because knowledge of binding time values is necessary to do the rewriting.

On the other hand it would be better if one could improve the process of partial evaluation in such a way that no rewriting is nessesary. Two methods generalizing the ideas used in partial evaluation look promising: Supercompilation [15] and Generalized Partial Computation [7].

7 Acknowledgement

Many people have contributed in various ways; but I would especially like to thank Anders Bondorf, Torben Mogensen, Olivier Danvy and Neil D. Jones for valuable inspiration, help and comments. I would also like to thank Jette Holm Broløs, Grethe Jørgensen and the members of the Topps group at DIKU.

References

[1] Lennart Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy, France. Lecture Notes in Computer Science 201*, pages 368–381, Springer-Verlag, 1985.

[2] 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.

[3] Anders Bondorf. Compiling laziness by partial evaluation. In [**?**], pages 9–22, 1991.

[4] Anders Bondorf and Olivier Danvy. *Automatic autoprojection of recursive equations with global variables and abstract data types.* Technical Report 90-4, DIKU, University of Copenhagen, Denmark, 1990.

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

[6] Pär Emanuelson. From abstract model to efficient compilation of patterns. In M. Dezani-Ciancaglini and U. Montanan, editors, *International Symposium on Programming, 5th Colloquium, Turin, Lecture Notes in Computer Science 137*, pages 91–104, Springer-Verlag, April 1982.

[7] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151, North-Holland, 1988.

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

[9] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE Computer Society 1990 International Conference on Computer Languages*, IEEE, March 1990.

[10] 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.

[11] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *Siam Journal on Computing*, 6(2):323–350, 1977.

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

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

[14] 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.

[15] Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

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

[17] Philip Wadler. Efficient compilation of pattern-matching. In Simon L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5, pages 78–103, Prentice-Hall, 1987.