Five

EFFICIENT COMPILATION OF PATTERN-MATCHING

Philip Wadler

This chapter shows how to compile function definitions with pattern-matching into case-expressions that can be efficiently evaluated. Previously, pattern-matching has been formally defined, and we have seen some examples of function definitions with pattern-matching.

5.1 Introduction and Examples

We begin by reviewing two examples.

The first example shows pattern-matching on more than one pattern. The function call (mappairs f xs ys) applies the function f to corresponding pairs from the lists xs and ys.

```
mappairs f [] ys        = []
mappairs f (x:xs) []    = []
mappairs f (x:xs) (y:ys) = f x y : mappairs f xs ys
```

For example, (mappairs (+) [1,2] [3,4]) returns [4,6]. The definition given here specifies that if the argument lists are not the same length, then the result will be as long as the shorter of the two lists. For example, (mappairs (+) [1,2] [3,4,5]) also returns [4,6].

The simplest way to think of pattern-matching is as trying to match each equation in turn. Within each equation, patterns are matched from left to right. For example, evaluating (mappairs (+) [1,2] [3,4] first matches (+) against f in the first equation, which succeeds, and then matches [1,2] against

[], which fails. Then the second equation is tried. Matching (+) against f and [1,2] against (x:xs) both succeed, but matching [3,4] against [] fails. Finally, matching in the third equation succeeds, binding f to (+), x to 1, xs to [2], y to 3, and ys to [4]. This corresponds exactly to the way pattern-matching was defined in Chapter 4.

Performing pattern-matching in this way can require a lot of work. The example above had to examine the list [1,2] three times and the list [3,4] twice. It seems clear that it should be possible to evaluate this function application in a more efficient manner that examines each list only once, but still gives the result prescribed by the semantics. This can be done by transforming the above definition into an equivalent one using case-expressions:

```
mappairs
= λf.λxs'.λys'.
    case xs' of
    NIL        => NIL
    CONS x xs  => case ys' of
                  NIL        => NIL
                  CONS y ys  => CONS (f x y) (mappairs f xs ys)
```

(Case-expressions were introduced in Section 4.4.) This chapter describes an algorithm that can automatically translate the first definition into the second. This algorithm is called the pattern-matching compiler.

The second example shows pattern-matching on a nested pattern. The function call (nodups xs) removes adjacent duplicate elements from a list xs. It can be defined as follows:

```
nodups []        = []
nodups [x]       = [x]
nodups (y:x:xs)  = nodups (x:xs),      y = x
                 = y : nodups (x:xs),  otherwise
```

(As you would expect, the guard 'otherwise' applies if no other guard does. See Appendix.) For example, (nodups [3,3,1,2,2,2,3]) returns [3,1,2,3]. Note that the naming need not be consistent: x stands for the first element of the list in the second equation, and for the second element of the list in the third equation.

Again, one can apply this definition by matching each equation in turn. For example, evaluation of (nodups [1,2,3]) will first try to match [1,2,3] against [], which fails. Next, it will try to match [1,2,3] against [x], which also fails. Finally, it will succeed in matching [1,2,3] against (y:x:xs), binding y to 1, x to 2 and xs to [3]. Again, this corresponds exactly to the semantics in Chapter 4.

As before, this is not very efficient. The list [1,2,3] is examined three times, and the sublist [2,3] is examined twice (once in the second equation, where it fails to match [], and once in the third equation, where it succeeds in matching (x:xs)). The pattern-matching compiler can transform this into a form that examines the list and the sublist only once:

```
nodups
= λxs''. case xs'' of
         NIL           => NIL
         CONS x' xs'   =>
                  case xs' of
                  NIL          =>  CONS x' NIL
                  CONS x xs    =>  IF (= x' x)
                                       (nodups (CONS x xs))
                                       (CONS x' (nodups (CONS x xs)))
```

(Here x' is the variable that was called x in the second equation and y in the third.)

The two kinds of pattern-matching, nested patterns and multiple patterns, are closely related to one another. The pattern-matching compiler discussed below works uniformly for both.

In the examples above, the patterns on the left-hand sides of the equations do not overlap. Many people would rewrite the first definition in the form:

```
mappairs' f [] ys        = []
mappairs' f xs []        = []
mappairs' f (x:xs) (y:ys) = f x y : mappairs' f xs ys
```

In this case, the patterns overlap because both the first and the second equation match against (mappairs' f [] []).

One reason for preferring mappairs' to mappairs is that it is considered to be more efficient. Indeed, if the simplest implementation of pattern-matching is used, matching each equation in turn, then it is slightly less work to match against xs than to match against (x:xs). However, as we shall see, this definition may actually be *less* efficient when the pattern-matching compiler is used. Some other problems with definitions like mappairs' will be discussed in Section 5.5.

The remainder of this chapter is organized as follows. Section 5.2 explains the pattern-matching compiler algorithm. Section 5.3 presents a Miranda program that implements the algorithm. Section 5.4 describes some optimizations to the pattern-matching compiler. Section 5.5 discusses a restricted class of definitions, called uniform definitions, which have useful properties.

Credit for the first published description of a pattern-matching compiler goes to Augustsson, who used it in the LML compiler [Augustsson, 1985]. Techniques similar to Augustsson's have been discovered independently by several researchers, including the authors of the Hope compiler [Burstall *et al.*, 1980]. The material presented here is derived partly from Augustsson's paper and partly from original work by the author (Wadler).

It is also possible to derive the pattern-matching compiler from its specification using program transformation techniques; see Barrett and Wadler [1986].

5.2 The Pattern-matching Compiler Algorithm

A Miranda function definition of the form

$$\begin{array}{l} \mathsf{f}\ p_{1,1}\ \ldots\ p_{1,n} = E_1 \\ \ldots \\ \mathsf{f}\ p_{m,1}\ \ldots\ p_{m,n} = E_m \end{array}$$

can be translated into the enriched lambda calculus definition

$$\begin{array}{ll} \mathsf{f} = \lambda u_1 \ldots \lambda u_n. & ((\lambda p_{1,1}'. \ \ldots\ \lambda p_{1,n}'.E_1')\ u_1\ \ldots\ u_n) \\ & [\!]\ \ldots \\ & [\!]\ ((\lambda p_{m,1}'. \ \ldots\ \lambda p_{m,n}'.E_m')\ u_1\ \ldots\ u_n) \\ & [\!]\ \mathsf{ERROR} \end{array}$$

where the u_i are new variables which do not occur free in any E_i, and the E_i' and $p_{i,j}'$ are the result of translating the E_i and $p_{i,j}$ respectively. It was shown how to do this translation in Chapter 4, using the **TD** translation scheme.

This section shows how to transform the definition of f into a form which uses case-expressions, removing all use of pattern-matching lambda abstractions. The transformation applies to the entire body of the $\lambda u_1 \ldots \lambda u_n$ abstraction, except that we generalize slightly to allow an arbitrary expression instead of ERROR.

For the sake of simplicity, we assume that constant patterns have been replaced by conditional equations, as described in Section 4.2.1.

5.2.1 The Function match

Our goal, then, is to transform an expression of the form

$$\begin{array}{l} \quad ((\lambda p_{1,1} \ldots \lambda p_{1,n}.E_1)\ u_1\ \ldots\ u_n) \\ [\!]\ \ldots \\ [\!]\ ((\lambda p_{m,1}. \ \ldots\ \lambda p_{m,n}.E_m)\ u_1\ \ldots\ u_n) \\ [\!]\ E \end{array} \tag{5.1}$$

into an equivalent expression which uses case-expressions rather than pattern-matching lambda abstractions.

The transformation is a bit complicated, and so we will use some new notation to describe it. Specifically, we will use a function **match**, which takes as its arguments the various parts of the input expression, namely the $p_{i,j}$, E_i and u_j, and produces as its output the transformed expression. The function **match** is similar to the **TD** and **TE** translation schemes introduced in Chapter 3, except that both its input and its result are enriched lambda calculus expressions. Furthermore, the double square bracket syntax becomes somewhat cumbersome, so we use a syntax like Miranda instead.

Here, then, is the call to **match** which we will use to compile the expression

(5.1) given above:

```
match [u₁, ..., uₙ]
      [( [p₁,₁, ..., p₁,ₙ], E₁ ),
       ...
       ( [pₘ,₁, ..., pₘ,ₙ], Eₘ )]
      E
```

This call should return an expression equivalent to the expression (5.1), and we take (5.1) as the *definition* of **match** from a semantic point of view. A call of **match** takes three arguments: a list of variables, a list of equations and a default expression. Each equation is a pair, consisting of a list of patterns (representing the left-hand side of the equation) and an expression (representing the right-hand side). Notice that the list of variables and each list of patterns have the same length.

We will also sometimes write calls of **match** in the form

```
match us qs E
```

Here us is the list of argument variables (of length n), and qs is a list of equations (of length m). Each equation q_i in qs has the form (ps_i, E_i), where ps_i is the list of patterns on the left-hand side (of length n) and E_i is the expression on the right-hand side.

As a running example, we will use the following Miranda function:

```
demo f [] ys          = A f ys
demo f (x:xs) []      = B f x xs
demo f (x:xs) (y:ys)  = C f x xs y ys
```

This function is similar in structure to mappairs, but it has been changed slightly in order to simplify and clarify the following examples. The right-hand sides use three unspecified expressions A, B and C.

Translating this into the enriched lambda calculus using TD gives:

```
demo
= λu₁.λu₂.λu₃. ((λf.λNIL.λys.A f ys) u₁ u₂ u₃)
               ▯ ((λf.λ(CONS x xs).λNIL.B f x xs) u₁ u₂ u₃)
               ▯ ((λf.λ(CONS x xs).λ(CONS y ys).C f x xs y ys)u₁ u₂ u₃)
               ▯ ERROR
```

where u_1, u_2, u_3 are new variable names which do not occur free in A, B or C. Now, we transform the definition of demo, by replacing its body with a call of **match**:

```
demo
= λu₁.λu₂.λu₃. match [u₁, u₂, u₃]
                     [ ( [f, NIL,          ys      ], (A f ys)        ),
                       ( [f, CONS x xs, NIL        ], (B f x xs)      ),
                       ( [f, CONS x xs, CONS y ys], (C f x xs y ys)) ]
                     ERROR
```

The following sections give rules to transform any call of **match** to an equivalent case-expression. We begin with rules for simple cases and proceed to more general cases.

5.2.2 The Variable Rule

In the example above, we have the following call on **match**:

```
match [u1, u2, u3]
      [ ( [f, NIL,              ys      ], (A f ys)          ),
        ( [f, CONS x xs, NIL           ], (B f x xs)        ),
        ( [f, CONS x xs, CONS y ys], (C f x xs y ys) ) ]
        ERROR
```

In this case, the list of patterns in every equation begins with a variable. This may be reduced to the equivalent call:

```
match [u2, u3]
      [ ( [NIL,           ys      ], (A u1 ys)          ),
        ( [CONS x xs, NIL         ], (B u1 x xs)        ),
        ( [CONS x xs, CONS y ys], (C u1 x xs y ys) ) ]
      ERROR
```

This is derived by removing the first variable, u_1, and in each equation removing the corresponding formal variable, f, and replacing f by u_1 in the right-hand side of each equation.

The same method works whenever each equation begins with a variable, even if each equation begins with a different variable. For example,

```
match [u2, u3]
      [ ( [x, NIL],          (B x) ),
        ( [y, CONS x xs], (C y x xs) ) ]
      ERROR
```

reduces to the call,

```
match [u3]
      [ ( [NIL],          (B u2) ),
        ( [CONS x xs],  (C u2 x xs) ) ]
      ERROR
```

(This particular example arises when compiling the definition of nodups.)

In general, if every equation begins with a variable pattern, then the call of **match** will have the form:

```
match (u:us)
      [ ( (v1:ps1), E1 ),
        . . .
        ( (vm:psm), Em ) ]
      E
```

This can be reduced to the equivalent call:

```
match us
      [ ( ps₁, E₁[u/v₁] ),
        . . .
        ( psₘ, Eₘ[u/vₘ] )]
      E
```

where, as usual, E[M/x] means 'E with M substituted for x'. In order to avoid too many subscripts, a Miranda-like notation has been used here; for example, we write (u:us) instead of $[u_1, \ldots, u_n]$. The general case corresponds to the first example above, where u is u_1, us is $[u_2, u_3]$, v_1 is f, ps_1 is [NIL, ys], and so on.

It is not hard to show that the rule is correct, that is, that the two **match** expressions are equivalent. This follows from the definition of **match** and the semantics of pattern-matching.

5.2.3 The Constructor Rule

The above step has left us with the following call of **match**:

```
match [u₂, u₃]
      [ ( [NIL,          ys         ], (A u₁ ys)          ),
        ( [CONS x xs, NIL           ], (B u₁ x xs)        ),
        ( [CONS x xs, CONS y ys], (C u₁ x xs y ys) ) ]
      ERROR
```

In this case, the list of patterns in every equation begins with a constructor. This call is equivalent to the following **case**-expression:

```
case u₂ of
    NIL          ⇒  match [u₃]
                          [ ( [ys],                (A u₁ ys)          ) ]
                          ERROR
    CONS u₄ u₅  ⇒  match [u₄, u₅, u₃]
                          [ ( [x, xs, NIL],          (B u₁ x xs)        ),
                            ( [x, xs, CONS y ys], (C u₁ x xs y ys) ) ]
                          ERROR
```

This call is derived by grouping together all equations that begin with the same constructor. Within each group, new variables are introduced corresponding to each field of the constructor. Thus NIL, which has no fields, requires no new variables, while CONS, which has two fields, introduces the variables u_4 and u_5. These new variables are matched against the corresponding subpatterns of the original patterns.

It may be useful here to look at a second example. In compiling the definition of a function like nodups, one would encounter the following call of

match:

```
match [u₁]
      [ ( [NIL],                   A          ),
        ( [CONS x NIL],            (B x)      ),
        ( [CONS y (CONS x xs)],    (C y x xs) ) ]
      ERROR
```

This can be reduced to the equivalent expression:

```
case u₁ of
    NIL         =>  match []
                          [ ( [],           A           ) ]
                          ERROR
    CONS u₂ u₃  =>  match [u₂, u₃]
                          [ ( [x, NIL],         (B x)       ),
                            ( [y, CONS x xs],   (C y x xs)  ) ]
                          ERROR
```

Again, NIL introduces no new variables (leaving a call of **match** with an empty list of variables), and CONS introduces two new variables, u_2 and u_3.

More generally, it may be the case that not all equations beginning with the same constructor appear next to each other. For example, one might have a call of **match** such as:

```
match [u₁]
      [ ( [CONS x NIL],            (B x)       ),
        ( [NIL],                   A           ),
        ( [CONS y (CONS x xs)],    (C y x xs)  ) ]
      ERROR
```

It is always safe to exchange two equations that begin with a different constructor, so we may rearrange the above to the equivalent call:

```
match [u₁]
      [ ( [NIL],                   A          ),
        ( [CONS x NIL],            (B x)      ),
        ( [CONS y (CONS x xs)],    (C y x xs) ) ]
      ERROR
```

which may be transformed as before.

It may also be the case that not all constructors appear in the original list of equations. For example, a function definition such as:

```
last [x]         = x
last (y:(x:xs))  = last (x:xs)
```

will result in the following call of **match**:

```
match [u₁]
      [ ( [CONS x NIL],            x                    ),
        ( [CONS y (CONS x xs)],    (last (CONS x xs))   ) ]
      ERROR
```

This can be reduced to the equivalent expression:

```
case u₁ of
     NIL            =>  match [] [] ERROR
     CONS u₂ u₃     =>  match [u₂, u₃]
                              [( [x, NIL],            x                   ),
                               ( [y, CONS x xs], (last (CONS x xs))  ) ]
                              ERROR
```

The case-expression must still contain a clause for the missing constructor, and the call of **match** in this clause will have an empty list of equations. (From the definition of **match**, we know that (**match** [] [] ERROR) is equivalent to ERROR.)

We now discuss the general rule for reducing a call of **match** where every equation begins with a constructor pattern. Say that the constructors are from a type which has constructors $c_1, \ldots, c_k$. Then the equations can be rearranged into groups of equations $qs_1, \ldots, qs_k$, such that every equation in group qs_i begins with constructor c_i. (If there is some constructor c_i that begins no equation, like NIL in the last example above, then the corresponding group qs_i will be empty.) The call of **match** will then have the form:

```
match (u:us) (qs₁ ++ ... ++ qsₖ) E
```

where each qs_i has the form:

$$[\ (\ ((c_i\ ps'_{i,1}):ps_{i,1}),\ E_{i,1}\ )$$
$$\ldots$$
$$(\ ((c_i\ ps'_{i,m_i}):ps_{i,m_i}),\ E_{i,m_i}\ )\ ]$$

(++ is list append.) In this expression we have abbreviated the constructor pattern ($c\ p_1 \ldots p_r$) to the form (c ps), where ps stands for the list of patterns $[p_1, p_2, \ldots, p_r]$. This call to **match** is reduced to the case-expression:

```
case u of
     c₁ us'₁  =>  match (us'₁ ++ us) qs'₁ E
     ...
     cₖ us'ₖ  =>  match (us'ₖ ++ us) qs'ₖ E
```

where each qs'_i has the form:

$$[\ (\ (ps'_{i,1}\ {+}{+}\ ps_{i,1}),\ E_{i,1}\ ),$$
$$\ldots$$
$$(\ (ps'_{i,m_i}\ {+}{+}\ ps_{i,m_i}),\ E_{i,m_i}\ )\ ]$$

Here each us'_i is a list of new variables, containing one variable for each field in c_i.

For instance, in the example at the beginning of this section, qs_2 is

```
[ ( [CONS x xs, NIL        ], (B u₁ x xs)        ),
  ( [CONS x xs, CONS y ys], (C u₁ x xs y ys) ) ]
```

and c_2 is CONS, $ps'_{2,1}$ is [x, xs], $ps_{2,1}$ is [NIL], $E_{2,1}$ is (B u_1 x xs), $ps'_{2,2}$ is [x, xs], $ps_{2,2}$ is [CONS y ys], and $E_{2,2}$ is (C u_1 x xs y ys). The corresponding qs'_2 is

```
[ ( [x, xs, NIL        ], (B u1 x xs)        ),
  ( [x, xs, CONS y ys], (C u1 x xs y ys) ) ]
```

The corresponding list of new variables, us_2', is [u_4,u_5].

This notation is, of necessity, rather clumsy. The reader will be pleased to discover, in Section 5.3, that this transformation can be written as a functional program which is more concise and (with experience) easier to read.

Again, the correctness of this rule can be proved using the definition of **match** and the semantics of pattern-matching.

5.2.4 The Empty Rule

After repeated application of the rules above, one eventually arrives at a call of **match** where the variable list is empty, such as the following:

```
match []
      [ ( [], (A u1 u3) ) ]
      ERROR
```

This reduces to:

```
(A u1 u3)
```

The correctness of this follows immediately from the definition of **match**, since A cannot return FAIL.

In general, the call of **match** may involve zero, one or more equations. Zero equations may result if the constructor rule is applied and some constructor of the type appears in no equations, as in **last** above. More than one equation can result if some of the original equations overlap.

Thus, the general form of a call of **match** with an empty variable list is:

```
match []
      [ ( [], E1 ),
        . . .
        ( [], Em ) ]
      E
```

where $m \geq 0$. From the definition of **match**, this reduces to

```
E1 [] ... [] Em [] E
```

Further, we can often guarantee that none of E_1, ..., E_m can be equal to FAIL. In this case, the above **match** expression reduces to E_1 if $m > 0$ and to E if $m = 0$. Section 5.4.2 discusses this optimization further.

5.2.5 An Example

The rules given so far are sufficient to translate the definitions of mappairs and nodups to the corresponding case-expressions given in the introduction. Notice that the variable names used in the introduction were chosen for readability. In practice, the translation algorithm will usually pick new names.

The reader may wish to verify that the rules given above are indeed sufficient to translate the definition

```
mappairs f [] ys      = []
mappairs f (x:xs) []  = []
mappairs f (x:xs) (y:ys) = f x y : mappairs f xs ys
```

to the equivalent:

```
mappairs
= λu1.λu2.λu3.
    case u2 of
    NIL        => NIL
    CONS u4 u5 => case u3 of
                  NIL        => NIL
                  CONS u6 u7 => CONS (u1 u4 u6)
                                     (mappairs u1 u5 u7)
```

The reader may also wish to check that the function nodups transforms to the case-expression given in the introduction.

5.2.6 The Mixture Rule

The above rules are sufficient for compiling most function definitions into case-expressions. However, there is still one case which has not been covered. This arises when not all equations begin with a variable, and not all equations begin with a constructor; that is, when there is a mixture of both kinds of equation. For example, here is an alternative definition of demo (similar in structure to the alternative definition of mappairs):

```
demo' f [] ys        = A f ys
demo' f xs []        = B f xs
demo' f (x:xs) (y:ys) = C f x xs y ys
```

Converting this to a **match** expression and applying the variable rule to eliminate f results in the following:

```
match [u2,u3]
      [ ( [NIL,          ys      ], (A u1 ys)          ),
        ( [xs,           NIL     ], (B u1 xs)          ),
        ( [CONS x xs, CONS y ys], (C u1 x xs y ys) ) ]
      ERROR
```

Neither the variable rule nor the constructor rule applies to this expression, because some equations begin with constructors and others with variables.

This is where the third argument to the **match** function is useful. The above expression is equivalent to:

```
match [u₂, u₃]
      [([NIL, ys], (A u₁ ys))]
      ( match [u₂, u₃]
              [([xs, NIL], (B u₁ xs))]
              ( match [u₂, u₃]
                      [([CONS x xs, CONS y ys], (C u₁ x xs y ys))]
                      ERROR ))
```

That is, the equations are broken into groups; first an equation beginning with a constructor, then one beginning with a variable, and then one beginning with a constructor again. If the equation in the first call of **match** fails to match the arguments then the value of the second call of **match** is returned. Similarly, if the equation in the second call does not match then the third call is returned, and if the equation in the third call does not match then ERROR is returned.

The reader may verify that reducing the three calls of **match** using the variable, constructor and base case rules results in the following definition of demo′:

```
demo′
= λu₁.λu₂.λu₃.
      case u₂ of
      NIL        ⇒ (A u₁ u₃)
      CONS u₄ u₅ ⇒
            case u₃ of
            NIL        ⇒ (B u₁ u₂)
            CONS u₆ u₇ ⇒
                  case u₂ of
                  NIL        ⇒ ERROR
                  CONS u₄ u₅ ⇒
                        case u₃ of
                        NIL        ⇒ ERROR
                        CONS u₆ u₇ ⇒ (C u₁ u₄ u₅ u₆ u₇)
```

This involves four case-expressions. When the second and third arguments are both non-empty lists then each list is examined twice, as compared with once for the definition of demo. This confirms the claim made in the introduction that 'optimizing' the definition of mappairs by transforming it into mappairs′ can actually result in worse code.

It may be possible to devise a compilation algorithm that would produce better code for this case. This could be done by simplifying a case-expression that appears inside another case-expression for the same variable. This sort of optimization is straightforward, although it requires considerably more book-keeping. In this case, mappairs′ would compile to the same case-expression as mappairs, although the compilation process would be rather more complicated.

In general, a call of **match** where some equations begin with variables and some with constructors may be transformed as follows. Say we are given a call of **match** of the form

```
match us qs E
```

The equation list qs may be partitioned into k lists $qs_1, \ldots, qs_k$ such that

```
qs = qs₁ ++ ... ++ qsₖ
```

The partition should be chosen so that each qs_i either has every equation beginning with a variable or every equation beginning with a constructor. (In the example above, each qs_i had length 1, but in general this need not be the case.) Then the call of **match** can be reduced to:

```
match us qs₁ (match us qs₂ ( ... (match us qsₖ E)...))
```

It is easy to use the definition of **match** to show that this rule is correct.

5.2.7 Completeness

With the addition of the mixture rule, it is now possible to reduce any possible call of **match** to a case-expression. This can be seen by a simple analysis. Given a call (**match** us qs E) then us will be either empty, so the empty rule applies, or non-empty. If us is non-empty then each equation must have a non-empty pattern list, which must begin with either a variable or a constructor. If all equations begin with a variable then the variable rule applies; if all begin with a constructor then the constructor rule applies; and if some begin with variables and some with constructors then the mixture rule applies.

Further, define the 'size' of an equation list as the sum of the sizes of all the patterns in the equation list. It can be seen that all four of the rules result in calls of **match** with smaller equation lists. This guarantees that the algorithm must eventually terminate.

5.3 The Pattern-matching Compiler in Miranda

This section presents the transformation algorithm as a functional program in Miranda.

5.3.1 Patterns

First, it is necessary to give a data type for representing patterns.

```
pattern      ::= VAR variable
               |    CON constructor [pattern]

variable     == [char]
constructor  == [char]
```

For example, (x:xs) is represented by (CON "CONS" [VAR "x", VAR "xs"]).

We need two functions on constructor names. The function arity given a constructor returns its arity, and the function constructors given a constructor returns a list of all constructors of its type:

```
arity        :: constructor -> num
constructors :: constructor -> [constructor]
```

For example (arity "NIL") returns 0, and (arity "CONS") returns 2. Both (constructors "NIL") and (constructors "CONS") return the list ["NIL", "CONS"].

5.3.2 Expressions

Next, we need a data type for representing expressions:

```
expression ::= CASE variable [clause]
               | FATBAR expression expression
               | ...
clause     ::= CLAUSE constructor [variable] expression
```

For example, the case-expression:

```
case xs of
    NIL         => E1
    CONS y ys   => E2
```

would be represented by

```
CASE "xs"
      [CLAUSE "NIL" [] E1',
       CLAUSE "CONS" ["y", "ys"] E2']
```

where E_1', E_2' are the representations of the expressions E_1, E_2. Similarly, the expression

E_1 [] E_2

would be represented by

FATBAR E_1' E_2'

The '. . .' in the definition of the type expression stands for other constructors used to represent other expressions, such as variables, applications and lambda abstractions. We do not need to know anything about these other expressions, except that there is a substitution function defined for them.

```
subst :: expression -> variable -> variable -> expression
```

For example, if E represents the expression (f x y), then (subst E "_u1" "x") represents the expression (f _u1 y).

5.3.3 Equations

An equation is a list of patterns paired with an expression:

```
equation == ([pattern], expression)
```

We will use the letter q to denote equations, or else write (ps,e).

We need functions to determine if an equation begins with a variable or a constructor. If it begins with a constructor, we also need a function to return that constructor.

```
isVar                       :: equation -> bool
isVar (VAR v      : ps, e)  = True
isVar (CON c ps' : ps, e)   = False

isCon                       :: equation -> bool
isCon q                     = ~ (isVar q)
getCon                      :: equation -> constructor
getCon (CON c ps' : ps, e)= c
```

5.3.4 Variable Names

We need some way of generating the new variable names, u1, u2, and so on. To do this we introduce a function makeVar that, given a number, returns a variable name.

```
makeVar   :: num -> variable
makeVar k = "_u" ++ show k
```

For example, (makeVar 3) returns '_u3'. Here we preface each new variable name with '_' to avoid it being confused with any variable already in the program.

5.3.5 The Functions partition and foldr

The implementation of the mixture rule uses a function called partition. The call (partition f xs) returns a list [xs_1, ..., xs_n] such that xs = xs_1 ++ ... ++ xs_n, and such that f x = f x' for any elements x and x' in xs_i, i from 1 to n, and such that f x $\neq$ f x' for any elements x in xs_i and x' in xs_{i+1}, i from 1 to n−1. For example,

```
partition odd [1,3,2,4,1] = [ [1,3], [2,4], [1] ]
```

The function partition is defined as follows:

```
partition              :: (* -> **) -> [*] -> [ [*] ]
partition f []         = []
partition f [x]        = [ [x] ]
partition f (x:x':xs)  = tack x (partition f (x':xs)),  f x = f x'
                       = [x] : partition f (x':xs),     otherwise

tack x xss             = (x : hd xss) : tl xss
```

Incidentally, the following definition of tack is *not* equivalent to the above definition:

```
tack x (xs:xss) = (x : xs) : xss
```

The difference between the two is closely related to the question of strict and lazy pattern-matching, mentioned in Section 4.3.5 in connection with the function firsts.

The pattern-matching compiler also uses the standard function foldr. The function foldr is defined so that

foldr f a [x_1, x_2, ..., x_n] = f x_1 (f x_2 (... (f x_n a)...))

For example, (foldr (+) 0 xs) returns the sum of the list of numbers xs. The function foldr is defined by:

```
foldr            :: (* -> ** -> **) -> ** -> [*] -> **
foldr f a []     = a
foldr f a (x:xs) = f x (foldr f a xs)
```

5.3.6 The Function match

We are now ready to define the function match. Calls of match have the form (match k us qs def). Here, as in Section 5.2, us represents a list of variables, qs represents a list of equations and def is a default expression. The argument k is added to help in generating new variable names; it should be chosen so that for every i>k, (makeVar i) is a new variable not in us, qs or def.

For example, the initial call to match to compile the definitions of mappairs would be:

```
match 3
      ["_u1", "_u2", "_u3"]
      [ ( [VAR "f", CON  "NIL" [],
                    VAR "ys"       ], E1 ),
        ( [VAR "f", CON "CONS" [VAR "x", VAR "xs"],
                    CON "NIL" [] ], E2 ),
        ( [VAR "f", CON "CONS" [VAR "x", VAR "xs"],
                    CON "CONS" [VAR "y", VAR "ys"] ], E3 ) ]
      error
```

where E_1, E_2 and E_3 represent the three expressions on the right-hand sides of the equation, and error represents the expression ERROR.

The definition of match can now be derived in a fairly straightforward way from the description given in Section 5.2. The type of match is:

```
match :: num -> [variable] -> [equation] -> expression -> expression
```

The equations for the top-level of match come from the empty rule and the mixture rule.

```
match k []     qs def = foldr FATBAR def [e | ([],e) <- qs ]
match k (u:us) qs def
  = foldr (matchVarCon k (u:us)) def (partition isVar qs)
```

The function matchVarCon is given a list of equations that either all begin with a variable or all begin with a constructor. It calls matchVar or matchCon, as appropriate.

```
matchVarCon k us qs def
 = matchVar k us qs def,        isVar (hd qs)
 = matchCon k us qs def,        isCon (hd qs)
```

The function matchVar implements the variable rule.

```
matchVar k (u:us) qs def
 = match k us [(ps, subst e u v) | (VAR v : ps, e) <- qs] def
```

The functions matchCon and matchClause implement the constructor rule. The call (choose c qs) returns all equations that begin with constructor c.

```
matchCon k (u:us) qs def
 = CASE u [matchClause c k (u:us) (choose c qs) def | c <- cs]
   where
   cs = constructors (getCon (hd qs))

matchClause c k (u:us) qs def
 = CLAUSE c us' (match (k'+k)
                       (us'++us)
                       [(ps'++ps, e) | (CON c ps' : ps, e) <- qs]
                       def )
   where
   k'  = arity c
   us' = [makeVar (i+k) | i <- [1..k'] ]

choose c qs = [q | q <- qs; getCon q = c]
```

This completes the Miranda program for the pattern-matching compiler.

5.4 Optimizations

This section discusses some optimizations to the pattern-matching compiler. Section 5.4.1 describes an optimization which gives greater efficiency when compiling overlapping equations. This involves further uses of [] and FAIL, and Section 5.4.2 describes how these may often be eliminated.

5.4.1 Case-expressions with Default Clauses

If overlapping equations are allowed, then sometimes the pattern-matching compiler described above may transform a small set of equations into a case-expression that is much larger. For example, consider the function defined by:

```
unwieldy [] [] = A
unwieldy xs ys = B xs ys
```

The pattern-matching compiler transforms this into:

```
unwieldy = λxs.λys. case xs of
                        NIL        => case ys of
                                        NIL         => A
                                        CONS y' ys' => B xs ys
                        CONS x' xs' => B xs ys
```

Here the expression (B xs ys) appears twice. If (B xs ys) were replaced by a very large expression, the increase in size caused by the compilation process could be very significant.

The problem can be avoided by modifying the rules given in Section 5.2 so that right-hand sides are never duplicated during the compilation process. In fact, only one rule can cause right-hand sides to be duplicated, the constructor rule. This rule is modified as follows.

Recall that the constructor rule transforms a call of **match** of the form:

```
match (u:us) (qs₁ ++ ... ++ qsₖ) E
```

to a case-expression of the form:

```
case u of
    c₁ us₁'  =>  match (us₁' ++ us) qs₁' E
    ...
    cₖ usₖ'  =>  match (usₖ' ++ us) qsₖ' E
```

where qs_1, ..., qs_k and qs_1', ..., qs_k' are as described in Section 5.2.3.

Normally E will be ERROR, but if the mixture rule is used then E may itself be a **match** expression containing right-hand sides; it is in this case that duplication may occur. The modified rule prevents this by using ▯ and FAIL to avoid duplicating E.

This is done by replacing the case-expression above with the equivalent expression:

```
(case u of
     c₁ us₁'  =>  match (us₁' ++ us) qs₁' FAIL
     ...
     cₖ usₖ'  =>  match (usₖ' ++ us) qsₖ' FAIL)
▯ E
```

If we call the old case-expression C, then the new expression is (C' ▯ E), where C' is formed by replacing each E in C by FAIL. It is clear that the new expression is equivalent to the old expression and, as desired, E is not duplicated by the new rule.

For example, using the new rule, the definition of unwieldy will now transform to:

```
unwieldy = λxs.λys.
              (case xs of
                  NIL         ⇒  (case ys of
                                       NIL          ⇒  A
                                       CONS y′ ys′  ⇒  FAIL) (a)
                                 [] FAIL                      (b)
                  CONS x′ xs′ ⇒  FAIL)
              [] B xs ys                                      (c)
```

This expression is a little larger than the previous version of unwieldy, but now (B xs ys) appears only once. If (B xs ys) stands for a large expression, then this new expression may be much smaller than the previous one.

As an example of how this sort of expression is evaluated, consider the call

```
(unwieldy NIL (CONS 1 NIL))
```

This is evaluated as follows. First, the outer case-expression is evaluated. Since xs is NIL, this causes the inner case to be evaluated. Since ys is (CONS 1 NIL), the inner case-expression returns FAIL; see line (a). So the expression after the inner [] is returned, which is also FAIL; see line (b). Thus, the outer case-expression returns FAIL. So the expression after the outer [] is returned; see line (c). This is (B NIL (CONS 1 NIL)), which is the value returned by the call of unwieldy.

5.4.2 Optimizing Expressions Containing [] and FAIL

It is often the case that all occurrences of FAIL, and its companion, [], can be eliminated. Most of these optimizations depend on reasoning that FAIL can never be returned by an expression, because in this case an occurrence of [] can be eliminated.

Suppose that FAIL is returned by an expression E. Then it is necessary (though not sufficient) that one of the following conditions must hold:

(i) FAIL is mentioned explicitly in E;
(ii) E contains a pattern-matching lambda abstraction, whose application may fail;
(iii) FAIL is the value of one of the free variables of E.

If the pattern-matching compiler described in this chapter is applied throughout, then no pattern-matching lambda abstractions will remain in the transformed program, and hence (ii) cannot occur. Since the programmer presumably cannot write FAIL explicitly in his program, it is not hard (although perhaps tedious) to verify that (iii) cannot occur either.

These observations focus our attention on all the places where FAIL can be introduced explicitly by the compiler. There are only two such places:

(i) In the translation of conditional equations (Section 4.2.6). Fortunately, we can easily transform conditional equations to avoid the use of [] and FAIL, and we show how to do so below.

(ii) In the variant of the pattern-matching compiler described in the last section, where the introduction of [] and FAIL seems unavoidable. This problem motivates the discussion in Section 5.5, in which we describe a restricted class of function definitions that can always be compiled without using [] and FAIL.

5.4.2.1 Rules for transforming [] and FAIL

We now give some rules for transforming expressions involving [] and FAIL to a simpler form. In all cases their correctness follows directly from the semantics of [].

First, we may eliminate [] if FAIL cannot occur on the left:

E_1 [] E_2 ≡ E_1

provided that E_1 cannot return FAIL.

For example, this rule is used to derive the optimized version of the empty rule in Section 5.2.4.

Second, we may eliminate [] if FAIL definitely occurs on the right or left:

E [] FAIL ≡ E and FAIL [] E ≡ E

For example, these rules can be used to simplify the final definition of unwieldy in Section 5.4.1.

Third, there is the following useful transformation involving IF:

(IF E_1 E_2 E_3) [] E ≡ IF E_1 E_2 (E_3 [] E)

provided that neither E_1 nor E_2 can return FAIL.

This rule will be useful in simplifying conditional equations, which we now attend to.

5.4.2.2 Eliminating [] and FAIL from conditional equations

The empty rule for **match**, which was described in Section 5.2.4, resulted in an expression of the form

E_1 [] ... [] E_m [] E

Now, the E_i are just the right-hand sides of the original equations. If a right-hand side consisted of a set of guarded alternatives without a final 'otherwise' case, then it will have been translated to the form:

IF G_1 A_1 (IF ... (IF G_g A_g FAIL) ...)

where g is the number of alternatives (see Section 4.2.6). If there was a final 'otherwise' case (that is, a final alternative with no guard, so that the right-hand side never fails), then it would have been translated to the form:

IF G_1 A_1 (IF ... (IF G_{g-1} A_{g-1} A_g) ...)

Notice that G_i and A_i cannot be equal to FAIL, because they are only the transformed versions of expressions written by the programmer.

If the right-hand side is of the first form, we can use the third rule of the previous section repeatedly, followed by the second, to give:

```
(IF G₁ A₁ (IF ... (IF Gg Ag FAIL) ... )) [] E
                ≡
IF G₁ A₁ (IF ... (IF Gg Ag E) ... )
```

If the right-hand side is of the second form, it cannot return FAIL, and so we can use the first rule of the previous section.

Application of these three rules will eliminate all occurrences of [] and FAIL in the expression generated by the empty rule, and incidentally thereby give a worthwhile improvement in efficiency.

5.4.2.3 Clever compilation

Using these rules, many of the instances of [] and FAIL remaining in a function definition can be eliminated. Later we will consider compiling an expression into low-level machine code. When we do this, we will see that it is possible to compile the remaining expressions involving [] and FAIL in a surprisingly efficient way, so that [] requires no code at all, and the FAIL simply compiles to a jump instruction. This is discussed in Section 20.4.

5.5 Uniform Definitions

This section introduces a restricted class of function definitions, called *uniform definitions*. There are two motivations for studying this class. First, uniform definitions avoid certain problems with reasoning about function definitions that involve pattern-matching. Second, uniform definitions are easier to compile, and are guaranteed to avoid certain kinds of inefficient code.

We begin by discussing some problems with reasoning about function definitions containing pattern-matching. Consider again the alternate definition of mappairs:

```
mappairs' f [] ys       = []
mappairs' f xs []       = []
mappairs' f (x:xs) (y:ys) = f x y : mappairs' f xs ys
```

Now, consider evaluation of the expression:

```
mappairs' (+) bottom []
```

where the evaluation of bottom would fail to terminate (for example, bottom could be defined by the degenerate equation bottom = bottom). Matching against the first equation binds f to (+) and then attempts to match [] against bottom. In order to perform this match it is necessary to evaluate bottom, and this of course causes the entire expression to fail to terminate.

On the other hand, consider evaluation of:

```
mappairs' (+) [] bottom
```

Now matching against the first equation binds f to (+), matching [] against [] succeeds, and then ys is bound to bottom (without evaluating bottom). So the expression returns [] instead of failing to terminate. This means that the definition of mappairs' is not as symmetric as it appears.

Further, if the first two equations of mappairs' were written in the opposite order, the two expressions above would change their meaning: now the first would return [] and the second would fail to terminate. So even though the first and second equations have the same right-hand side, the order in which they are written is important.

The original definition of mappairs has none of these problems:

```
mappairs f [] ys         = []
mappairs f (x:xs) []     = []
mappairs f (x:xs) (y:ys) = f x y : mappairs f xs ys
```

Now the asymmetry between (mappairs (+) [] bottom) and (mappairs (+) bottom []) is apparent from the equations. Further, changing the order of the equations does not change the meaning of the function.

In general, one might expect that whenever the equations do not overlap, the order in which they are written does not matter. In fact, this is not true. Consider the definition:

```
diagonal x     True  False = 1
diagonal False y     True  = 2
diagonal True  False z     = 3
```

The three equations of this definition are non-overlapping, that is, at most one equation can apply. However, by this definition, the evaluation of:

```
diagonal bottom True False
```

would return 1. On the other hand, if the order of equations in the definition were reversed, so the third equation came first, then the above expression would fail to terminate. So even though the equations do not overlap, the order in which they are written is important.

Clearly, it would be useful to have a test that guarantees that the order of the equations does not matter. We now define the class of *uniform* definitions, which have this property. The definition of 'uniformity' is designed so that it is easy to test whether a definition is uniform while applying the pattern-matching compiler to it.

DEFINITION

A set of equations is *uniform* if one of the following three conditions holds:

(i) either, all equations begin with a variable pattern, and applying the variable rule (of Section 5.2.2) yields a new set of equations that is also uniform;
(ii) or, all equations begin with a constructor pattern, and applying the constructor rule (of Section 5.2.3) yields new sets of equations that are all also uniform;
(iii) or, all equations have an empty list of patterns, so the empty rule (of Section 5.2.4) applies, and there is at most one equation in the set.

That is, a set of equations is uniform if it can be compiled without using the mixture rule (of Section 5.2.6), and if the empty rule is only applied to sets containing zero or one equations. (It is easy for the reader to check that when the empty rule is applied to more than one equation, the order is relevant.)

Such equation sets are called 'uniform' because all equations must begin the same way, either with a variable pattern or a constructor pattern, whereas the mixture rule applies when some equations begin with variable patterns and some with constructor patterns.

It is not difficult to prove the following:

THEOREM

If a definition is uniform, changing the order of the equations does not change the meaning of the definition.

The proof is a straightforward induction, and is similar in structure to the proof of correctness of the pattern-matching compiler that was outlined (along with its definition) in Section 5.2.

This shows that being uniform is a sufficient condition for the order of the equations not to matter. It is not a necessary condition, as is shown by the function dummy:

```
dummy [] = 1
dummy xs = 1, xs = []
```

Clearly, dummy is not uniform, but the order of the equations does not matter. However, the following result shows that being uniform is indeed necessary if one considers only the left-hand sides:

THEOREM

If the left-hand sides of a definition are such that the order of the equations does not matter (regardless of the right-hand sides or condition parts of the equations), the definition is uniform.

For example, the order of the equations would matter in dummy if the 1 in the second equation were changed to a 2. Again, the proof of the theorem is a straightforward induction. These two theorems give us a simpler way of characterizing uniform equations, without referring to the pattern-matching compiler. Namely, a definition is uniform if and only if its left-hand sides are such that the order of the equations does not matter.

It is also possible to show that every uniform definition is non-overlapping. The converse is not true: the function diagonal is non-overlapping but is not uniform. Researchers have often referred to 'lack of overlapping' as an important property, but perhaps they should refer to 'uniformity' instead, since this is the property that guarantees that the order of equations does not matter.

Uniform equations are related to *strongly left-sequential equations* as defined by Hoffman and O'Donnell [1983], which are in turn related to *sequential equations* as defined by Huet and Levy [1979].

Notice that although uniform equations are independent of 'top-to-bottom' order, they still have a 'left-to-right' bias. For example, although the following definition is uniform:

```
xor False x     = x
xor True  False = True
xor True  True  = False
```

the same definition with the arguments interchanged is not:

```
xor' x     False = x
xor' False True  = True
xor' True  True  = False
```

Of course, we can always get around this bias by using extra definitions to rearrange the arguments. For example, we can define

```
xor'' x y = xor y x
```

and then xor'' is equivalent to xor', and both xor'' and xor have uniform definitions.

The existence of left-to-right bias is due to the semantics of pattern-matching that we have chosen. A different definition of pattern-matching that avoids left-to-right bias is possible; see Huet and Levy [1979].

There is a second reason why uniform equations are important: they are easier to implement. The problems with implementing non-uniform definitions have been referred to implicitly in previous sections. In summary, they are as follows:

(i) The resulting case-expressions may examine some variables more than once (see Section 5.2.6).
(ii) The compiler must use a modified constructor rule to avoid duplicating the right-hand side of equations (see Section 5.4.1).

(iii) The resulting expressions may contain [] and FAIL. Implementing such expressions efficiently requires additional simplification rules and/or a special way of implementing FAIL using jump instructions (see Section 5.4.2).

The result is that the pattern-matching compiler must be significantly more complicated if it is to deal with non-uniform expressions. Further, the first point above means that it may be difficult to know how efficient the code compiled for a non-uniform definition will be.

An issue related to uniformity is the way conditionals are handled. In languages such as SASL, conditional expressions and where expressions may appear anywhere in an expression, and the semantics of each is defined independently. In Miranda, conditions and where clauses are not separate expressions, but rather must be associated with the right-hand side of definitions. This increases the power of Miranda, in some ways, but only when non-uniform definitions are used. Hence, a restriction to uniform equations would also allow this part of the language to be simplified.

On the other hand, it should be pointed out that non-uniform definitions are sometimes very convenient. For example, the following definition reverses lists of length two, and leaves all other lists the same:

```
reverseTwo [x,y] = [y,x]
reverseTwo xs    = xs
```

The most straightforward way of rewriting this as a uniform definition is much more long-winded:

```
reverseTwo []         = []
reverseTwo [x]        = [x]
reverseTwo [x,y]      = [y,x]
reverseTwo (x:y:z:ws) = x:y:z:ws
```

In this case, it is easy to see another way of rewriting reverseTwo, but, in general, rewriting may not be so easy.

Functional language designers have long debated whether or not definitions with overlapping equations should be allowed in functional languages. As has been shown, it may be more appropriate to debate the merits of uniform – as opposed to non-overlapping – equations. Several arguments in favor of restricting definitions to uniform equations have been raised here; but it is also true that non-uniform definitions are on occasion quite convenient. No doubt the debate will continue to be a lively one.

*   *   *

Acknowledgements: For help with my work on chapters 4, 5 and 7 I would like to thank the following. Simon Finn made valuable comments, both detailed and deep. Simon Peyton Jones and I are listed as co-authors only on Chapter 4, but he has provided as much thoughtful help as a co-author on the other chapters as well. For help of a very different nature, I thank Catherine Lyons.

This work was performed while on a research fellowship supported by ICL.

References

Augustsson, L. 1985. Compiling pattern matching. In *Conference on Functional Programming Languages and Computer Architecture, Nancy*, pp. 368–81. Jouannaud (editor), LNCS 201. Springer Verlag.

Barrett, G., and Wadler, P. 1986. *Derivation of a Pattern-matching Compiler.* Programming Research Group, Oxford.

Burstall, R.M., MacQueen, D.B., and Sanella, D.T. 1980. HOPE: an experimental applicative language. In *Proceedings of the ACM Lisp Conference.* August.

Hoffmann, C.M., and O'Donnell, M.J. 1983. Implementation of an interpreter for abstract equations. In *10th ACM Symposium on Principles of Programming Languages*, pp. 111–21. ACM.

Huet, G., and Levy, J.J. 1979. *Computations in Non-ambiguous Linear Term Rewriting Systems.* INRIA technical report 359.