Derivation of a Pattern-Matching Compiler

Geoff Barrett and Philip Wadler
Oxford University Computing Laboratory, Programming Research Group

1986

Introduction

This paper presents the derivation of an efficient compiler for pattern-matching in a functional language. The algorithm is described in [A] and [W], and is used in the LML compiler [J]. The derivation is based on that given in [Bar].

The algorithm relates to functional languages in two ways: a functional language is used as the medium of the derivation, and the algorithm is useful for compiling functional languages. The features of functional languages that make them well suited to program transformation, such as referential transparency, are widely known [BD]. Less widely known is the fact that it is now possible to write efficient compilers for such languages [P], and indeed the LML compiler is itself written in LML.

The derivation relies heavily on the use of higher-order functions, such as *fold* and *map*, that encapsulate common patterns of computation, and on the laws satisfied by these functions. The higher-order functions and their associated laws are well-known and have applications in a wide variety of contexts: for instance, see [Bu], [Bi], and [BW]. This paper provides an extended example of how these laws can be used to derive a practical and significant result. The derivation also relies heavily on familiar mathematical properties, such as associativity, distributivity, and commutativity.

These results are encouraging, because they suggest that much of the hard work of many derivations can be factored into a few general theorems, and that many of the concepts and manipulations necessary for such derivations will already be familiar from traditional algebra.

The emergence of general laws and an algebraic framework indicate how the field of program derivation has matured. In early work on the transformation of functional programs many similar derivations were repeated to derive similar results [BD, F]. Experience with such derivations has led to the emergence of more general theorems, and many derivations can be structured and shortened by the use of such theorems.

The value of program derivations most often quoted is that they increase the reliability of the resulting program. Another equally important value is that they lead to deeper understanding. Knowledge of the derivation given here was very useful in writing the explanation of the algorithm given in [W]. In particular, the structure of the explanation there follows closely the structure of the derivation here.

1 Notation

The notation used for structures and functions is essentially that of SQUIGOL (see [Bi]). The list constructor is the infix (:) and signatures are introduced by (::). All infix operators have the same precedence and associate to the right. Four standard functions are used throughout, namely $*$ for map, $=$ for folding functions which are associative and have an identity, $\Rrightarrow$ for folding functions which are not associative or have no identity and $==$ which constructs the list of corresponding pairs from two lists. Their definitions follow:

$$\begin{aligned} f * [\,] &= [\,] \\ f * (a : x) &= f\ a : f * x \end{aligned}$$

$$\begin{aligned} f \Rrightarrow a\ [\,] &= a \\ f \Rrightarrow a\ (b : x) &= f\ b\,(f \Rrightarrow a\,x) \end{aligned}$$

$$(f{=}) = f \Rrightarrow e \quad \textbf{where} \quad e \text{ is the unit of } f$$

$$\begin{aligned} [\,] == [\,] &= [\,] \\ (a : x) == (b : y) &= (a; b) : x == y \end{aligned}$$

Two particular functions are used throughout: $+\!\!+$ appends two lists; *flatten* produces a list from a list of lists:

$$x +\!\!+ y = (:) \Rrightarrow y\ x$$

$$\mathit{flatten} = ((+\!\!+){=})$$

1.1 Specification of the function *group*

The derivation will require a function *group* which groups the elements of a list into segments on which a function, f, is constant. These segments are tagged with the value of f on the segment. For instance,

$$\begin{aligned} \mathit{group}\ f\ \ [4; 0; 3; 5; 6] \quad = \quad & [(\mathit{false}; [4])\ ;\ (\mathit{true}; [0; 3])\ ;\ (\mathit{false}; [5; 6])] \\ & \textbf{where} \quad f\ x = x \leq 3 \end{aligned}$$

We can specify the situation where y is a grouping of x under f as follows:

$$\begin{aligned} \mathit{groups}\ f\ x\ y \iff \quad & \mathit{flatten}(\mathit{snd} * y) = x \\ & \qquad\qquad \wedge \\ & (\wedge) = [f\ a = c \mid (c; z) \leftarrow y; a \leftarrow z] \end{aligned}$$

The *group* function is uniquely specified by taking the shortest possible grouping of x:

$$\mathit{groups}\ f\ x\ \ (\mathit{group}\ f\ x) \wedge \forall\ y \mid y\ \mathit{groups}\ f\ x\ y \cdot\ \#(\mathit{group}\ f\ x) \leq \#y$$

It can easily be shown that $\mathit{snd}{*}(\mathit{group}\ f\ x)$ contains no empty lists. (Otherwise, removing elements of the form $(a; [\,])$ retains the grouping property but is shorter.)

A possible implementation of *group* is as follows:

$$\begin{array}{lcl} group\ f\ x & = & group' \Rrightarrow [\,]\ (graph\ f * x) \\ & & \textbf{where}\quad graph\ f\ a = (f\ a; a) \\ \\ group'\ [\,]\ fbb & = & [fbb] \\ group'\ ((fa; x) : g)\ (fb; b) & = & \begin{cases} (fa; b : x) : g; & fa = fb \\ (fb; [b]) : (fa; x) : g; & fa \neq fb \end{cases} \end{array}$$

2 An Abstract Language for Pattern Matching

In this section, we introduce a language in which the relevant features of pattern matching may be expressed. The language is designed to obey a number of elegant, algebraic laws by means of which it is given a semantics. The abstract syntax of this language is given in terms of operators on the various types.

2.1 Syntax

Three types are primitive to the discussion. The type of variable names *var*. The type of constructors *con*; with each constructor there is associated a natural number, called its *arity* :: $con \longrightarrow nat$, which gives the maximum number of terms to which the constructor may be applied. For example, the arity of the empty list is 0 and the arity of the list constructor is 2. The third type is that of expressions which we shall introduce presently.

¿From the *var* and *con* types, we construct the type of patterns. A pattern is either a variable or a constructor with a list of sub-patterns whose length is the arity of the constructor:

$$pat ::= \mathbf{v}\ var \mid \mathbf{c}\ con\ [pat]$$

where $\mathbf{c}\ c\ ps$ is well-formed just when $arity\ c = \#ps$.

We now come to the type of expressions, *exp*. We require the type to have the operation of variable renaming:

$$sub :: (var; var) \longrightarrow exp \longrightarrow exp$$

where $ren\ (u; v)\ e$ denotes the renaming of u for v in e; further, we assume a case expression so that we have a function

$$\textbf{case}\ \_\ \textbf{of}\ \_ :: var \longrightarrow [(pat; exp)] \longrightarrow exp$$

and the following pattern-matching operators:

$$\begin{array}{c} \mathbf{e} :: exp \\ \mathbf{f} :: exp \\ \_ \Rightarrow \_ :: (var; pat) \longrightarrow exp \longrightarrow exp \\ \_ \,\square\, \_ :: exp \longrightarrow exp \longrightarrow exp \end{array}$$

where **e** is the value returned when a complete set of equations has failed to match; **f** is the exception returned when a single equation does not apply.

The phrase $(v; p) \Rightarrow e$ matches the binding of v against p. If the match succeeds then the appropriate subterms of the binding are bound to the variables of p and e is evaluated in the new environment; otherwise, the whole phrase is equivalent to **f**. It is well-formed only when the variables of p are not bound in e and v is not free in e. The bound variables of the phrase are the bound variables of e and the variables appearing in p; the free variables are v along with the free variables of e which do not appear in p.

The phrase $e_1 \,[\!]\, e_2$ denotes the successive attempts to evaluate the matches e_1 and e_2. If the first succeeds, then the second is discarded; otherwise, the second match is tried. The phrase is always well-formed; its free variables are the free variables of e_1 and the free variables of e_2; likewise the bound variables.

2.2 Semantics

The pattern-matching language is given a semantics by means of a set of algebraic equivalences. The intended equivalence is the largest in which the constants **e** and **f** are not equivalent, the operators $\_ \,[\!]\, \_$ and $(v; p) \Rightarrow \_$ and congruences and the following laws are true.

We first give the semantics of $(v; p) \Rightarrow \_$. This involves describing the way in which variables are bound and expressions are matched against patterns. Matching against a variable pattern always succeeds. The effect is to rename the variable in the final expression:

$$(u; \mathbf{v}\ v) \Rightarrow e \equiv ren\ (u; v)\ e \tag{1}$$

Under suitable disjointness conditions on bound variables, $(u; \mathbf{v}\ v) \Rightarrow \_$ and $(u; p) \Rightarrow \_$ commute:

$$(u; \mathbf{v}\ v) \Rightarrow (u'; p) \Rightarrow e \equiv (u'; p) \Rightarrow (u; \mathbf{v}\ v) \Rightarrow e \tag{2}$$

whenever $v \neq u'$ and u is not a variable of p. Note that by the well-formedness conditions, v may not be a variable of p and since u' is free in the expression $(u; \mathbf{v}\ v) \Rightarrow (u'; p) \Rightarrow e$, $u \neq u'$. For the sake of completeness we consider the case in which $v = u'$. In this case, the binding of u is bound to v and so we need only match u against p for $u' = v$ may not be free in e by the well-formedness rules:

$$(u; \mathbf{v}\ v) \Rightarrow (v; p) \Rightarrow e \equiv (u; p) \Rightarrow e$$

Note that these rules are not independent.

When matching a constructor against a pattern, the matching may be performed one constructor at a time so that the sub-terms of the binding of u which matches the outermost constructor of the pattern may be bound to new variables which are then matched against the sub-patterns:

$$(v; \mathbf{c}\ c\ ps) \Rightarrow e \equiv (v; \mathbf{c}\ c\ \ (\mathbf{v} * us)) \Rightarrow (\Rightarrow) \Rrightarrow e\ \ (us == ps) \tag{3}$$

where us is a list of distinct variables which do not appear in any of the ps and are not free in e; and $\#us = arity\ c$.

All we need to know of the operator $\_ \,[]\, \_$ is that it is associative and has identity $\mathbf{f}$:

$$\begin{array}{rcl} e_1 \,[]\, (e_2 \,[]\, e_3) & \equiv & (e_1 \,[]\, e_2) \,[]\, e_3 \\ \mathbf{f} \,[]\, e & \equiv & e \\ e \,[]\, \mathbf{f} & \equiv & e \end{array} \tag{4}$$

It is not in general true that $[]$ is commutative. Consider, for instance, that u is bound to $\perp$ in the match

$$((u; \mathbf{c}\ c\ ps) \Rightarrow e_1) \,[]\, ((u; \mathbf{v}\ v) \Rightarrow e_2)$$

This phrase must be equivalent to $\perp$. However, the phrase

$$((u; \mathbf{v}\ v) \Rightarrow e_2) \,[]\, ((u; \mathbf{c}\ c\ ps) \Rightarrow e_1)$$

is equivalent to $ren\ (u; v)\ e$, which may not be $\perp$. Attempting to match against patterns with distinct constructors does commute since everything except $\perp$ fails to match at least one of them:

$$((v; \mathbf{c}\ c_1\ ps_1) \Rightarrow e_1 \,[]\, (v; \mathbf{c}\ c_2\ ps_2) \Rightarrow e_2) \equiv ((v; \mathbf{c}\ c_2\ ps_2) \Rightarrow e_2 \,[]\, (v; \mathbf{c}\ c_1\ ps_1) \Rightarrow e_1)\ \{c \neq c'\} \tag{5}$$

The last law is a distributivity law:

$$(v; p) \Rightarrow (e_1 \,[]\, e_2) \equiv ((v; p) \Rightarrow e_1) \,[]\, ((v; p) \Rightarrow e_2) \tag{6}$$

which holds because if the match $(v; p) \Rightarrow$ fails then both sides are equivalent to $\mathbf{f}$; if the match succeeds, evaluating $e_1 \,[]\, e_2$ in an environment which binds the variables of p is the same as evaluating e_1 and e_2 in that environment and choosing the first to succeed.

2.3 Equations and Case Expressions

Concretely, a named list of equations looks like:

$$\begin{array}{l} mappairs\ f\ [\,]\ ys\ \ \ \ \ \ \ \ \ \ \ \ \ \ = [\,] \\ mappairs\ f\ (x : xs)\ [\,]\ \ \ \ \ \ \ \ = [\,] \\ mappairs\ f\ (x : xs)\ (y : ys) = f\ x\ y : mappairs\ f\ xs\ ys \end{array}$$

An equation is nothing more than a list of patterns followed by an expression. For example, the first equation of the function *mappairs* is

$$([\mathbf{v}\ f; \mathbf{c}\ NIL\ [\,]\,; \mathbf{v}\ ys]\,; NIL)$$

so that in general we define

$$equn == ([pat]\,; exp)$$

Furthermore, if *mappairs* is applied to the terms u_1, u_2 and u_3, and the first equation applies, then the result is given by

$$(u_1; \mathbf{v}\ f) \Rightarrow (u_2; \mathbf{c}\ NIL\ [\,]) \Rightarrow (u_3; \mathbf{v}\ ys) \Rightarrow NIL$$

and, in general, given a list of variables, *us*, the meaning of an equation is given by:

$$\begin{aligned} &\mathit{clause} :: [\mathit{var}] \longrightarrow \mathit{equn} \longrightarrow \mathit{match} \\ &\mathit{clause}\ \mathit{us}\ (\mathit{ps}; e) = (\Rightarrow) \Rrightarrow e\ (\mathit{us} == \mathit{ps}) \end{aligned}$$

A list of equations is intended to be tried in order from top to bottom so that if the first fails, the second is tried and so on. However, if no equation matches, the result is an error. We restrict ourselves to non-empty lists of equations whose pattern lists are all of the same length so that

$$\mathit{qs} :: \mathit{equns} == [\mathit{equn}]$$

is well-formed only when it is non-empty and

$$\mathit{all}\,[m = n \mid n \leftarrow \mathit{lens}]\ \textbf{where}\quad m : \mathit{lens} = [\#\mathit{ps} \mid (\mathit{ps}; e) \leftarrow \mathit{qs}]$$

If *mappairs* is applied to terms u_1, u_2 and u_3, the desired result is given by:

$$\begin{aligned} &(u_1; \mathbf{v}\ f) \Rightarrow (u_2; \mathbf{c}\ \mathit{NIL}\ [\,]) \Rightarrow (u_3; \mathbf{v}\ \mathit{ys}) \Rightarrow \mathit{NIL} \\ \square\ &(u_1; \mathbf{v}\ f) \Rightarrow (u_2; \mathbf{c}\ \mathit{CONS}\ [\mathbf{v}\ x; \mathbf{v}\ \mathit{xs}]) \Rightarrow (u_3; \mathbf{c}\ \mathit{NIL}\ [\,]) \Rightarrow \mathit{NIL} \\ \square\ &(u_1; \mathbf{v}\ f) \Rightarrow (u_2; \mathbf{c}\ \mathit{CONS}\ [\mathbf{v}\ x; \mathbf{v}\ \mathit{xs}]) \Rightarrow (u_3; \mathbf{c}\ \mathit{CONS}\ [\mathbf{v}\ y; \mathbf{v}\ \mathit{ys}]) \Rightarrow \\ &\qquad\qquad (f\ x\ y : \mathit{mappairs}\ f\ \mathit{xs}\ \mathit{ys}) \\ \square\ &\mathbf{e} \end{aligned} \tag{7}$$

so that **e** is returned if no equation matches. In general, the meaning of a set of equations is given by:

$$\begin{aligned} &\mathit{mapEqun} :: [\mathit{var}] \longrightarrow [\mathit{equn}] \longrightarrow \mathit{match} \\ &\mathit{mapEqun}\ \mathit{us}\ \mathit{qs} = (\square) = (\mathit{clause}\ \mathit{us}\ ) * \mathit{qs}\ \square\ \mathbf{e} \end{aligned}$$

where *us* are distinct, do not appear in the pattern of *qs* and $\#\mathit{us} = \#\mathit{ps}$ for $(\mathit{ps}; e) : \mathit{qs}' = \mathit{qs}$.

We will wish to compile this expression into case expressions. The meaning of a case expression is given in terms of $\Rightarrow$ and $\square$ as follows:

$$\begin{aligned} &\mathit{mapCase} :: \mathit{exp} \longrightarrow \mathit{exp} \\ &\mathit{mapCase}\,(\mathbf{case}\ v\ \mathbf{of}\ \mathit{pes}) = (\square) = (f * \mathit{pes}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{where}\quad f\ (p; e) = (v; p) \Rightarrow e \end{aligned}$$

Note that if no pattern is matched, the whole expression is equivalent to **f**.

3 The Problem

Consider an evaluation of expression (7), when u_1 is bound to $(+)$, u_2 is bound to $[1; 2]$ and u_3 is bound to $[0]$. The evaluation proceeds thus:

1. $(+)$ matches f; $(1 : [2])$ does not match $[\,]$;
2. $(+)$ matches f; $(1 : [2])$ matches $(x : \mathit{xs})$; $(0 : [\,])$ does not match $[\,]$;

3. (+) matches f; $(1 : [2])$ matches $(x : xs)$; $(0 : [\,])$ matches $(y : ys)$.

By using the laws (4), (6), (2), and (1), and the semantics of case expressions, the reader is invited to verify that the expression for $mappairs\ u_1\ u_2\ u_3$, (7), may be transformed to:

$$
\begin{array}{l}
\textbf{case}\ u_2\ \textbf{of} \\
\quad \mathbf{c}\ NIL\ [\,] \Rightarrow [\,] \\
\quad \mathbf{c}\ CONS\ [\mathbf{v}\ x; \mathbf{v}\ xs] \Rightarrow \\
\qquad \textbf{case}\ u_3\ \textbf{of} \\
\qquad\quad \mathbf{c}\ NIL\ [\,] \Rightarrow [\,] \\
\qquad\quad \mathbf{c}\ CONS\ [\mathbf{v}\ y; \mathbf{v}\ ys] \Rightarrow (u_1\ x\ y : mappairs\ u_1\ xs\ ys) \\
[\,]\ \mathbf{e}
\end{array}
$$

An evaluation of this yields:

1. $1 : [2]$ does not match $\mathbf{c}\ NIL\ [\,]$.
2. $1 : [2]$ matches $\mathbf{c}\ CONS\ [\mathbf{v}\ x; \mathbf{v}\ xs]$,
   (a) $0 : [\,]$ does not match $\mathbf{c}\ NIL\ [\,]$.
   (b) $0 : [\,]$ does match $\mathbf{c}\ CONS\ [\mathbf{v}\ y; \mathbf{v}\ ys]$.

In the first evaluation there are three successful matches followed by failure; in the second there are none.

This is the required output from the compiler. It has a number of features which serve as the specification of the problem. The most important and most obvious is that the output of the compiler means the same as its input. Secondly, all the patterns which are to be matched have only variables as sub-patterns. Furthermore, each case expression has a different constructor in each of its patterns.

The meaning-preserving part of the specification is formulated as:

$$
\begin{array}{l}
compile :: [var] \longrightarrow [equn] \longrightarrow exp \\
compile\ us\ qs\ [\,]\ \mathbf{e} \equiv mapEqun\ qs\ us
\end{array}
$$

Since this is a consequence of:

$$compile\ us\ qs \equiv ([\,]) = (clause\ us) * qs$$

we may take the latter as the specification of the compilation function.

4 The Derivation

The algorithm is derived in a number of cases. The first case depends upon whether are any patterns in the equations.

4.1 The degenerate Case

If there are no patterns in the list of equations, then it must be the case that the list of variables is empty:

$compile\ [\,]\ qs$

Thus, the definition is:

$$\boxed{compile\ [\,]\ qs = (\square) = snd * qs}$$

4.2 The non-degenerate case

When the list of patterns in each equation is non-empty, we start the compilation by grouping the equations into segments whose first patterns are variables or contain constructors. The two sorts of group require different compilation strategies; the correct strategy for each group is selected by the function *varconsCompile*.

The first function to be defined inspects the first pattern to see whether it is a variable or a constructor:

$$\boxed{\begin{array}{l} varcon :: equn \longrightarrow (\textsc{var} \mid \textsc{con}) \\ varcon\ (\mathbf{v}\ v : ps; m) \quad = \textsc{var} \\ varcon\ (\mathbf{c}\ c\ ps : ps'; m) = \textsc{con} \end{array}}$$

The result of grouping a list of equations according to this function will be a list of pairs, the first of which is VAR or CON. The function *varconsCompile* will select the correct strategy for each group but the required meaning can be given regardless of the compilation strategy so that the specification of *varconsCompile* is:

$$\begin{array}{l} varconsCompile :: [var] \longrightarrow ((\textsc{var} \mid \textsc{con})\,; [equn]) \longrightarrow exp \\ varconsCompile\ us\ (vc; qs) \equiv (\square) = clause\ us * qs \end{array}$$

Let $gqs = group\ varcon\ qs$ so that

qs

then we have

$compile\ (u : us)\ qs$

and this last line suffices for the definition of *compile*, so that:

$$\boxed{\begin{array}{l} compile\ (u : us)\ qs = \quad (\square) = varconsCompile\ (u : us) * gqs \\ \qquad\qquad\qquad\qquad\quad \mathbf{where} \quad gqs = group\ varcon\ qs \end{array}}$$

In order to define *varconsCompile*, we invent two new functions, *varCompile* and *consCompile* which embody the different compilation strategies. Each applies to a list of equations, but they have preconditions that the first pattern of each of the equations must be a variable or contain a constructor, respectively, so that the specifications are:

$$\begin{array}{lcl} varCompile; consCompile :: [var] \longrightarrow [equn] \longrightarrow exp \\ varCompile\ us\ qs & \equiv & (\!|\,|\!) = clause\ us * qs \\ consCompile\ us\ qs & \equiv & (\!|\,|\!) = clause\ us * qs \end{array}$$

and we can see that the following definition of *varconsCompile* meets its specification:

$$\boxed{\begin{array}{l} varconsCompile\ us\ (\textsc{var}; qs) = varCompile\ us\ qs \\ varconsCompile\ us\ (\textsc{con}; qs) = consCompile\ us\ qs \end{array}}$$

4.2.1 When all first patterns are variables

We begin to find a definition for *varCompile* by studying an application of *clause* to an equation whose first pattern is a variable:

$clause\ (u : us)\ (\mathbf{v}\ v : ps; e)$

Defining

$$\boxed{equnSubst\ u\ (\mathbf{v}\ v : ps; e) = (ps; ren\ (u; v)\ e)}$$

we have

$clause\ (u : us)\ (\mathbf{v}\ v : ps; e)$

Since the precondition of *varCompile* is that all first patterns must be variables, this reformulation of *clause* may be substituted in the specification:

$varCompile\ (u : us)\ qs$

Hence we can make the definition:

$$\boxed{varCompile\ (u : us)\ qs = compile\ us\ (equnSubst\ u * qs)}$$

4.2.2 When all first patterns have constructors

We now come to the compilation for a list of equations whose first pattern has a constructor. The aim is to use the commutativity rule, (5), in order to sort the equations so that equations whose first patterns contain the same constructor are adjacent. We first define a subsidiary order:

$$(u; \mathbf{c}\ c\ ps) \Rightarrow m \prec' (u; \mathbf{c}\ c'\ ps') \Rightarrow m' \Longleftrightarrow c \between c'$$

which is useful in the derivation, and a similar order on *equn* which will be used in the program:

$$\boxed{(\mathbf{c}\ c_1\ ps_1 : ps_1'; m_1) \prec (\mathbf{c}\ c_2\ ps_2 : ps_2'; m_2) \Longleftrightarrow c_1\ ¡\ c_2}$$

consCompile us qs

Let $oqs = stablesort\ (\prec)\ qs$. We next wish to group the equations according to the of the first pattern, so we define:

$$\boxed{\begin{array}{l} conName :: equn \longrightarrow con \\ conName\ (\mathbf{c}\ c\ ps'; e) = c \end{array}}$$

and let $gqs = group\ conName\ oqs$ so that:

oqs

Lastly, each group of equations is to form one branch of a case expression. The function which compiles the group is called *onConsCompile* and satisfies the specification:

$$\begin{array}{l} oneConsCompile :: [var] \longrightarrow (con; [equn]) \\ f\ u\ (oneConsCompile\ us\ (c; qs)) \equiv (\square) = clause\ (u : us)\ qs \\ \mathbf{where} \quad f\ u\ (p; e) = (u; p) \Rightarrow e \end{array}$$

so we have:

consCompile $(u : us)\ qs$

Hence we make the definition:

$$\boxed{\begin{array}{ll} consCompile\ us\ qs = & \mathbf{case}\ u\ \mathbf{of}\ oneConsCompile\ us * group\ conName\ oqs \\ & \mathbf{where} \quad oqs = stablesort\ (\prec)\ qs \end{array}}$$

4.2.3 When all first patterns have the same constructor

As in the case when all first patterns were variables, we begin by considering the application of *clause* to a single equation. The following lemma is useful in the derivation:

Lemma 1

$$clause\ (u : us)\ (p : ps; e) = (u; p) \Rightarrow clause\ us\ (ps; e)$$

Proof

clause $(u : us)\ (p : ps; e)$

□

The derivation proceeds:

clause $(u : us)\ (\mathbf{c}\ c\ ps' : ps; e)$

Defining

$$\boxed{\mathit{deconstruct}\ (\mathbf{c}\ c\ ps' : ps; e) = (ps' +\!\!+ ps; e)}$$

and letting $p' = \mathbf{c}\ c\ (\mathbf{v} * us')$, we have:

$\mathit{clause}\ (u : us)\ (\mathbf{c}\ c\ ps' : ps; e)$

Let $(p; e) = \mathit{oneConsCompile}\ us\ (c; qs)$, then

$(u; p) \Rightarrow e$

Hence we make the definition

$$\boxed{\mathit{oneConsCompile}\ us\ (c; qs) = (\mathbf{c}\ c\ (\mathbf{v} * us')\ ;\mathit{compile}\ (us' +\!\!+ us)\ (\mathit{deconstruct} * qs))}$$

4.3 The complete algorithm and a note on termination

Taking together all the definitions derived so far gives the final program:

$$\begin{array}{l}
\mathit{compile}\ [\,]\ qs \qquad\quad = (\square) = \mathit{snd} * qs \\
\mathit{compile}\ (u : us)\ qs = (\square) = \mathit{varconsCompile}\ (u : us) * \mathit{gqs} \\
\qquad\qquad\qquad\qquad\quad \textbf{where} \quad \mathit{gqs} = \mathit{group}\ \mathit{varcon}\ qs \\
\mathit{varcon}\ (\mathbf{v}\ v : ps; m) \quad = \mathit{VAR} \\
\mathit{varcon}\ (\mathbf{c}\ c\ ps : ps'; m) = \mathit{CON} \\
\\
\mathit{varconsCompile}\ us\ (\mathit{VAR}; qs) = \mathit{varCompile}\ us\ qs \\
\mathit{varconsCompile}\ us\ (\mathit{CON}; qs) = \mathit{consCompile}\ us\ qs \\
\\
\mathit{equnSubst}\ u\ (\mathbf{v}\ v : ps; e) = (ps; \mathit{ren}\ (u; v)\ e) \\
\\
\mathit{varCompile}\ (u : us)\ qs = \mathit{compile}\ us\ (\mathit{equnSubst}\ u * qs) \\
\\
(\mathbf{c}\ c_1\ ps_1 : ps_1'; m_1) \prec (\mathbf{c}\ c_2\ ps_2 : ps_2'; m_2) = c_1\ \mathbin{¡}\ c_2 \\
\\
\mathit{conName}\ (\mathbf{c}\ c\ ps'; e) = c \\
\\
\mathit{consCompile}\ us\ qs = \textbf{case}\ u\ \textbf{of}\ \mathit{oneConsCompile}\ us * \mathit{group}\ \mathit{conName}\ \mathit{oqs} \\
\qquad\qquad\qquad\qquad\quad \textbf{where} \quad \mathit{oqs} = \mathit{stablesort}\ (\prec)\ qs \\
\mathit{deconstruct}\ (\mathbf{c}\ c\ ps' : ps; e) = (ps' +\!\!+ ps; e) \\
\\
\mathit{oneConsCompile}\ us\ (c; qs) = (\mathbf{c}\ c\ (\mathbf{v} * us')\ ;\mathit{compile}\ (us' +\!\!+ us)\ (\mathit{deconstruct} * qs))
\end{array}$$

where us' is a list of new variables.

Defining the variant to be the sum of the arities of the constructors plus the number of constructors appearing in qs plus the length of us, the variant is reduced each time *compile* is applied. Thus, the algorithm terminates.

5 Conclusions

Although the basic laws of $\Box$ and $\Rightarrow$ and those pertaining to $=$ and $*$ are intuitively simple, the complete algorithm is not immediately comprehensible. The proof of correctness is an aid not only to why the compiler works but also to how it works. By abstracting away from the basic, unfamiliar operators in *exp* and relying on their more familiar properties, irrelevant details can be ignored and the proof becomes much simpler. In [Bar], this strategy is not employed and the derivation is at once far less comprehensible and four times longer!

The use of $=$ and $*$ to augment binary operators is widely applicable and generally produces well-modularised proofs. (See [Bi], for instance.) The derivation presented here contains no induction on lists. All of this manipulation can be factored into general theorems, and used again in other derivations.

The particular properties of associativity, commutativity and distributivity appear in other transformations (see [RH] where $\Box$ corresponds to *IF* or *ALT* and $\Rightarrow$ to *SEQ*, and [Bac] in which the transformation of non-deterministic to deterministic state-machines relies on identical properties for $|$ and $\cdot$). Programs are made more efficient by eliminating redundant calculations. One common method of achieving this is by accumulation of a result; in this transformation and in [Bac], however, the increase in efficiency relies on a distributivity property. The object of both the latter transformations is a decision procedure which is optimised by eliminating backtracking. In [RH] too, a normal form for finite programs is produced by, essentially, transformation into a decision procedure. Accumulation is an effective strategy when the predominant form of a program is sequential calculation; distribution applies when the predominant form is a conditional construct. This suggests that a 'distribution' strategy is a valuable member of the transformer's tool kit.

References

[A] Augustsson, L. Compiling Pattern Matching. *IFIP international conference on Functional Programming and Computer Architecture.* Nancy, September 1985.

[Bac] Backhouse, R. *The Syntax of Programming Languages.* Prentice Hall, 1979.

[Bar] Barrett, G. *Compiling Pattern Matching.* M.Sc. Dissertation, Oxford University, 1985.

[Bi] Bird, R.S. An introduction to the theory of lists. *International Summer School on Logic of Programming and Calculi of Discrete Design*, Marktoberdorf, Germany, August 1986.

[BW] Bird, R.S., and Wadler, P.L., *An Introduction to Functional Programming*, Prentice-Hall, 1988.

[Bu] Burge, W.H, *Recursive Programming Techniques.* Addison-Wesley, 1975.

[BD] Burstall, R.M. and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM*, **24**(1), January 1977.

[F] Feather, M.S. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, **4**(1):1–20, January 1982.

[J] Johnsson, T. Efficient compilation of lazy evaluation. *ACM SIGPLAN Symposium on Compiler Construction*, *SIGPLAN Notices*, **19**(6), June 1984.

[P] Peyton-Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[RH] Roscoe, A.W. and Hoare, C.A.R. *The Laws of occam Programming.* TCS 60 (1988).

[W] Wadler, P.L., Compiling pattern matching, in [P].

A Laws about Lists

The derivation depends on the following results about functions of lists. Most of these theorems are well-known and have straightforward proofs. For further discussion of similar laws, see [Bi] or [BW].

Theorem 1

$$(f*) \cdot (g*) = ((f \cdot g)*)$$

Theorem 2

$$\#x = \#y \Longrightarrow (x == y) +\!\!+ (u == v) = (x +\!\!+ u) == (y +\!\!+ v)$$

Theorem 3

$$((\oplus) =) \cdot \mathit{flatten} = ((\oplus) =) \cdot (((\oplus) =)*)$$

Theorem 4

$$(f*) \cdot \mathit{flatten} = \mathit{flatten} \cdot (((f*))*)$$

Theorem 5

$$(\oplus) \Rrightarrow ((\oplus) \Rrightarrow a\ x)\ \ y = (\oplus) \Rrightarrow a\ \ (y +\!\!+ x)$$

Theorem 6

$$(\forall\, a\, b\cdot\ f\ (a \oplus b) = f\ a \otimes f\ b) \Longrightarrow f\ ((\oplus) = (a : x)) = (\otimes) = (f * (a : x))$$

Theorem 7 If $\prec$ is a linear pre-order then

$$(a \prec b \Rightarrow a \oplus b = b \oplus a) \Longrightarrow ((\oplus) =) = ((\oplus) =) \cdot \mathit{stablesort}\ (\prec)$$

Theorem 8 If f is strictly monotonic then sorting and mapping f on a list commute:

$$(f\ a \prec' f\ b \Leftrightarrow a \prec b) \Longrightarrow \mathit{stablesort}\ (\prec') \cdot (f*) = (f*) \cdot \mathit{stablesort}\ (\prec)$$

Theorem 9

$$(\forall\, a\, b\cdot\ f\ (b \oplus a) = b \oplus f\ a) \Longrightarrow f \cdot (\oplus) \Rrightarrow a = (\oplus) \Rrightarrow (f\ a)$$