

Compiling Pattern Matching

Lennart Augustsson

Programming Methodology Group

Department of Computer Science

Chalmers University of Technology

S-412 96 Göteborg, Sweden

Email: augustss@chalmers.uucp or augustss@chalmers.csnet

Introduction

Pattern matching is a very powerful and useful device in programming. In functional languages it emerged in **SASL** [Turn76] and **Hope** [Burs80], and has also found its way into **SML** [Miln84]. The pattern matching described here is that of **LML** which is a lazy ([Frie76] and [Hend76]) variant of **ML**. The pattern matching in **LML** evolved independently of that in **SML** so they are not (yet) the same, although very similar. The compilation of pattern matching in **SML** has been addressed in [Card84].

The **LML** compiler project began as an attempt to produce efficient code for a typed functional language with lazy evaluation. Since we regard pattern matching as an important language feature it should also yield efficient code.

Only pattern matching in case expressions is described here, since we regard this as the basic pattern matching facility in the language. All other types of pattern matching used in **LML** can be easily translated into case expressions, see [Augu84] for details.

The compilation (of pattern matching) proceeds in several steps:

- transform all pattern matching to case expressions.
- transform complex case expressions into expressions that are easy to generate code for.
- generate G-code for the case expressions, and from that machine code for the target machine.

Type definitions

In **LML** all types (except the function type) are considered to be built up in the same way. A type is defined by a type definition in which all the constructors of the type are given. These constructors are the only elements of the type. For convenience some types are predefined (int, bool, list, char...) and there are special syntactic constructions to make it easier to use them (like the **if** expression), but they could equally well have been defined by the user. (There is one exception: the integer type is handled especially for efficiency reasons.)

A new type in **LML** is introduced by a type definition:

```

let type T( $v_1, \dots$ ) =  $C_1(t_{11}, \dots) + \dots + C_n(t_{n1}, \dots)$ 
in ...

```

Here T is the name of the new type, the v_i are type variables (if the new type is polymorphic), and the C_i are the (new) constructors of this type. The types of the arguments to the constructors are given by the type expressions t_{ij} (which may contain v_i).

Some examples:

```

let type bool = false() + true() in ...
let type rec list(*a) = nil() + cons(*a, list(*a)) in ...

```

Nullary constructors are normally written without parenthesis, e.g. `false`, but in this paper we will often use the parenthesis to avoid any confusion between variables and constructors. Here `*a` is a polymorphic type variable, for details see [Miln78].

As soon as a new type has been introduced the constructors of this type can be used to form expressions. An expression with a constructor as its outermost part is in canonical form, i.e. it yields itself as value when evaluated.

The only way to scrutinize a value of a type is with the **case** expression, which has the following syntax:

```

case e in
    p1 : e1
  ||   p2 : e2
  ...
  ||   pn : en
end

```

where the p_i are patterns and the e_i are expressions.

A pattern is built up from constructors (introduced by type definitions) and from variables. A pattern can be arbitrarily complex, i.e. the subparts of a constructor pattern may themselves be patterns.

A special class of patterns are the *simple patterns*. A simple pattern is either a variable or a constructor where all the subpatterns are variables. The simple patterns are especially interesting since a case expression consisting of only simple patterns can be compiled easily into very efficient code (described later).

E.g.

```

let last l =
    case l in
        cons(h, nil) : h
      ||   cons(h, t) : last t
    end
in ...

```

For convenience two other types of patterns may occur. The first is a wildcard pattern (written '\$'), which has the same meaning as a variable (i.e. it matches anything) except that it may not occur in the expression and it may be repeated in a pattern (which a ordinary variable may not be). The other

pattern is the **as** construction, which is useful when both the whole value and its subparts are needed.

Examples:

```
let hd (cons(a,$)) = a in ...
let hd_and_whole (cons(a,$) as l) = (a, l)
```

Both these constructions can be viewed as purely "syntactical sugar" and we will not consider them in the rest of this paper.

Case evaluation

The value of a case expression is equal to the value of the expression part of the first matching pattern. The patterns are checked from top to bottom, and each pattern is checked from left to right. The discrimination expression (the one after **case**) is evaluated exactly as much as is needed to determine which pattern is the first to match. If no pattern matches the value of the case expression is undefined (it will cause a runtime error).

The patterns are allowed to overlap, but an earlier pattern is not allowed to completely overlap a later one completely since then the later pattern could never match.

The expression part of the pattern is evaluated with all the variables in the pattern bound to the corresponding parts of the discrimination expression.

The explicit top-down ordering is necessary since patterns are allowed to overlap, if this was not the case the pattern order would be unimportant.

The left-right ordering is (unfortunately) needed when using lazy evaluation since it affects the termination properties of the program. A strict case construction would evaluate the discrimination expression completely, and if there were non-terminating parts it would not matter in which order the evaluation of the subparts would proceed since the result would not terminate anyway. When using lazy evaluation it is a little more complicated. The discrimination expression should only be evaluated enough to determine which pattern is the first to match, but when matching subparts of a pattern this must be done in some order. Ideally this order should not matter, but this would force parallel matching of the subparts.

The patterns of a case expression should be exhaustive, if they are not the compiler issues an error message and inserts a default entry. If this default entry is ever evaluated it will produce an error message and terminate the program.

An example

Before giving the algorithm, we will illustrate the ideas by means of how the algorithm works on a particular example.

The basic idea is to merge all patterns with the same outermost constructor and replace all the subpatterns with variables. These subpatterns must then be matched in new case expressions, which are treated in the same way.

The LML expression (in the concrete LML syntax)

```

case e in
  [x;true] : x
|| [false] : true
|| z : false
end
  corresponds to the following when using the abstract syntax for the patterns
case e in
  cons(x, cons(true(), nil())) : x
|| cons(false(), nil()) : true()
|| z : false()
end

```

This is transformed by joining the two `cons` patterns into one and by introducing new variables (see below). These new variables are then examined by further `case` expressions. When the new `case` expressions are introduced they may be non-exhaustive. This is not an error (as it would be on the top level), but instead an indication that the pattern (or patterns) that is expanded into that particular `case` expression does not match. Since overlapping patterns are allowed there may be other patterns that can match. The only overlap that can occur, when the patterns are simple, is that a variable overlaps a constructor (since equal constructors have already been merged). This means that if a failure to match occurs all the overlapping patterns should be checked. In the code this is indicated by the `default` expression. This is not an expression that is available to the user; it is only used during transformation. The value of the `default` expression is the same as that of the nearest surrounding default pattern (a default pattern is a pattern consisting of a single variable).

In the example below the first `default` expression indicates that the pattern `'cons(false(), nil())'` does not match, but since the `'false()'` part is overlapped by the subpattern `'x'` in `'cons(x, cons(true(), nil()))'` the expression may still match this. The value of default is therefore the same as that for the `'v3'` pattern which tests for a match on `'cons(true(), nil())'`. This may also fail (the second `default`), but the expression may still match since both pattern are overlapped by `'z'`. The second default is therefore equivalent to the value of that entry.

The `default` entries are easily compiled into jump instructions that just continue the execution at the appropriate default entry.

```

case e in
  cons(x, v2) :
    case x in
      false() :
        case v2 in
          nil() : true()
          || $ : default
        end
      || v3 : case v2 in
        cons(true(), nil()) : x
        || $ : default
      end
    end
  || z : false()
end

```

The new case expressions introduced by this transformation are now treated in the same way.

We then proceed in the same way with the new case expressions that are not simple and when all the complex patterns are removed it expands to

```

case e in
  cons(x, v2) :
    case x in
      false() :
        case v2 in
          nil() : true()
          || $ : default
        end
      || v3 : case v2 in
        cons(v4,v5) :
          case v4 in
            true() :
              case v5 in
                nil() : x
                || $ : default
              end
            || $ : default
          end
        || $ : default
      end
    || z : false()
  end

```

As can be seen from this example the transformed expression is often much larger than the original. This is not a serious problem in practice since most patterns used in real programs are simple or almost simple already. Also the transformed expressions will not be much larger when the patterns do not overlap, which is often the case.

Conditional Patterns

The patterns matching may extended further in that there may be a boolean expression associated with a pattern. In order for the pattern to match this expression must evaluate to true. This feature is present in KRC [Turn82] and is there called conditional equations. We will not here discuss how this can be compiled, but the algorithm presented here can be extended in a fairly simple way to allow such conditional patterns. The idea is that a pattern of the form

p & cond : e

would match only if the pattern **p** matches and the boolean expression **cond** evaluates to true (the condition can of course contain the variables bound by the pattern **p**). It would be transformed into something like

p : if cond then e else default

Conditional patterns would make it possible to compile a pattern with repeated occurrences of a variable. This would then be equivalent to a pattern with distinct variables and a condition expressing that they should be equal.

Transformation algorithm

The transformation algorithm can now be presented more formally.

The pattern expansion algorithm is called C , and is used in the following way

$$C \left(\langle e_1, e_2, \dots, e_n \rangle, \{ \langle p_{21}, p_{22}, \dots, p_{2n2} \rangle : E_2 \}, d \right) \\ \dots \\ \langle p_{m1}, p_{m2}, \dots, p_{mnm} \rangle : E_m$$

C takes as its first argument a sequence of expressions, and as its second argument a sequence of pairs. Each pair consists of a pattern sequence and an expression. The result of the transformation is an LML-expression which will give the value E_k if the pattern sequence $\langle p_{k1}, p_{k2}, \dots, p_{kn} \rangle$ is the first sequence that matches the expression sequence $\langle e_1, e_2, \dots, e_n \rangle$. E_k will be evaluated with all the variables in the patterns bound to the corresponding parts of the expressions. A sequence of patterns matches a sequence of expressions if each pattern matches the corresponding expression. The matching is checked from left to right. The parameter d is the value that should be returned if none of the pattern sequences match the expression sequence (this parameter is usually **default**).

The expression returned by C contains only simple patterns.

The expressions in the expression sequence are always variables except on the top level of the transformation (as can be seen below).

The arguments to C must have certain properties for it to work:

- the expressions and the patterns must be well typed.
- the patterns must not be completely overlapping (as described above).
- the length of all pattern sequences and the expression sequence must be equal (these properties are, of course preserved in recursive calls of C).

To transform the case expression

```
case e in
  p1 : e1
||  p2 : e2
...
||  pn : en
end
```

C is called with the arguments

$$C \left(\langle e \rangle, \{ \langle p_1 \rangle : e_1 \}, fail \right) \\ \dots \\ \langle p_n \rangle : e_n$$

which as a result would give an expression with the right meaning, since the one element

expression sequence is matched against the one element pattern sequences. The expression *fail* is an error value indicating that the pattern matching failed (this cannot happen when the patterns are exhaustive exhaustive).

C is elaborated by case analysis of the arguments. When recursive calls are made the arguments are always "simpler"; the decreasing measure used here is the maximum of the sums of the number of constructors and variables in each pattern sequence. This ensures the termination of C .

Case 1:

$$C(<>, \{<>:e\}, d)$$

The expression sequence is empty. This implies that all the pattern sequences are also empty since they always have the same length. The sequence of pairs must be a singleton, otherwise the patterns would be completely overlapping and this is not allowed. Since an empty sequence of patterns matches an empty sequence of expressions the result is simply

e

Case 2:

$$C(<e_1, e_2, \dots, e_n>, \{<v_1, p_{12}, \dots, p_{1n}>:E_1, \\ \dots \\ <v_m, p_{m2}, \dots, p_{mn}>:E_m\}, d)$$

A nonempty expression sequence, and the first pattern in each sequence is a variable.

Since a variable matches any expression this implies that v_k should be bound to e_1 when evaluating E_k . This is achieved by returning the expression

```

case  $e_1$  in
     $<p_{12}, \dots, p_{1n}>:E_1 [^v/v_1]$ 
   $v: C(<e_2, \dots, e_n>, \{<p_{22}, \dots, p_{2n}>:E_2 [^v/v_2]\}, d)$ 
     $\dots$ 
     $<p_{m2}, \dots, p_{mn}>:E_m [^v/v_m]$ 
end
  where  $v$  is a new unique variable.
  (This could also be written let  $v = e_1$  in  $C \dots$ )

```

That is, v is bound to e_1 when evaluating E_k , but all (free) occurrences of v_k in E_k have been replaced by v , which is equivalent to having v_k bound to e_1 in E_k .

Another solution would be to substitute e_1 for the v_k in the expressions and this would not normally be inefficient since the expression sequence normally consists of variables (as stated earlier).

The recursive call is obviously simpler since a variable has been removed from each pattern sequence.

Case 3:

$$C (<e_1, e_2, \dots, e_n>, \{ \dots \\ <C_1(q_{11}, \dots, q_{1n_1}), p_{12}, \dots, p_{1n}>:E_1 \\ <C_k(q_{k1}, \dots, q_{kn_k}), p_{k2}, \dots, p_{kn}>:E_k \\ <v_{k+1}, p_{k+1,2}, \dots, p_{k+1,n}>:E_{k+1} \\ \dots \\ <v_m, p_{m2}, \dots, p_{mn}>:E_m$$

A nonempty expression sequence, and the first pattern in each sequence is a constructor (called C_k and with subpatterns q_{k1}, \dots, q_{kn_k}), or a variable (called v_k). The first pattern sequences are constructor patterns and they may be followed by a number of variable patterns.

The pair sequence is first grouped by the constructor in the first pattern (C_j). The grouping consists of rearranging the pair sequence so that all those with the same leading constructor are put together and those with a leading variable form a separate group. Pattern order does not matter when the patterns are not overlapping, but since we have overlapping patterns the relative order of the sequences in the same group must not change. The relative order between different groups does not matter since they cannot overlap (because the leading constructors differ), except that the those with leading variables must come last. All this is easily achieved by having an ordering on the constructors and using the constructors as the key in a stable sort.

After the grouping we would have something like:

$$C (<e_1, e_2, \dots, e_n>, \{ \dots \\ <C^1(q_{11}, \dots, q_{1n_1}), p_{12}, \dots, p_{1n}>:E_1 \\ \dots \\ <C^1(q_{k1}, \dots, q_{kn_1}), p_{k2}, \dots, p_{kn}>:E_k \\ <C^N(q_{m1}, \dots, q_{mn_m}), p_{m2}, \dots, p_{mn}>:E_m \\ \dots \\ <v_j, p_{j2}, \dots, p_{jn}>:E_j$$

where C^1 is the first constructor and so on, C^N the last.

(The names of the subpatterns and variables are not the same as above for purely clerical reasons - it would be too messy with even more indicies.)

The result of this is

case e_i **in**

$$\begin{aligned}
 C^1(v_{11}, \dots, v_{1n_1}) & : C(<v_{11}, \dots, v_{1n_1}, e_2, \dots, e_n>, \{ <q_{k1}, \dots, q_{kj1}, p_{k2}, \dots, p_{kn}> : E_k \}, \text{default}) \\
 & \dots \\
 || \dots \\
 || C^N(v_{j1}, \dots, v_{jn_j}) & : C(<v_{j1}, \dots, v_{jn_j}, e_2, \dots, e_n>, \dots \\
 & \dots \\
 v & : C(<e_2, \dots, e_n>, \{ <p_{j2}, \dots, p_{jn}> : E_j [^v/v_j] \}, d) \\
 & \dots
 \end{aligned}$$

end

where all v_{ij} and v are new unique variables.

If there are no pattern sequences with a leading variable the last case entry (the default entry) would be replaced by

$v : d$

This has all the needed properties:

- the number of elements in the expression and pattern sequences in all recursive calls to C are the same since there is a new variable for each subpattern (for a particular constructor) and these are added to the expression sequence and the same number of patterns are added to the pattern sequences. (The number of subpatterns for a particular constructor must be the same in all instances since the expression is well typed).
- the result only contains simple patterns since C only returns simple patterns and we have introduced only simple patterns.
- the recursive call is made with simpler arguments (which guarantees termination). In each pattern sequence we have taken away a constructor pattern and replaced it by its subpatterns, which reduces the number of constructors.
- the returned expression also has the required behaviour since the outermost part of first expression is what is matched first and the recursive call will then give an expression which will match all the subparts of this and after that the other expressions. If no constructor matches the default entry is used. If there were any variable patterns they are treated exactly as in case 2. If there are no variable patterns it means that if the expression doesn't match any of the constructors it cannot match at all. A failure to match at this level should return what is indicated by the d parameter (which in most of the cases is a **default**, indicating that the surrounding default entry should be used).

Case 4:

$$\begin{aligned}
 C(<e_1, e_2, \dots, e_n>, \{ & \dots \\
 & <C_1(q_{11}, \dots, q_{1n_1}), p_{12}, \dots, p_{1n}> : E_1 \\
 & <v_k, p_{k2}, \dots, p_{kn}> : E_k \\
 & \dots \\
 & <C_m(q_{m1}, \dots, q_{mn_m}), p_{m2}, \dots, p_{mn}> : E_m
 \end{aligned}
), d)$$

A nonempty expression sequence, and the first pattern in each sequence is a constructor (called C_k and with subpatterns q_{k1}, \dots, q_{knk}), or a variable (called v_k). This case differs from the previous case, in that the constructor and variable patterns may appear in any order.

This case cannot be handled as easily as case 3 since there are variable patterns in between the constructor patterns. This means that the pattern sequences cannot be grouped as described above since this could alter the order of overlapping patterns. (Consider for instance the two patterns ' $C(x, 2)$ ' and ' $C(1, y)$ ' which if processed as above would be reversed and the value ' $C(1, 2)$ ' would then match the wrong pattern.)

Several approaches can be used to transform this:

- rearrange the pattern sequences so that case 3 can be used, but when two overlapping patterns sequences must change place a new pattern sequence is introduced - the one corresponding to the overlap between the patterns. This pattern is then placed before the two patterns changing that are places.
- divide the pair sequence into subsequences by repeatedly taking the longest prefix of it for which case 3 can be used. We then get a number of "sets" for which case 3 applies. If the first of those sequences fail to match the second one should be tried and so on. This can be accomplished by giving the expression for the second set as the default parameter when transforming the first one and so on.

We illustrate the second approach here. Let P_k be the subsequences of the pair sequence for which case 3 can be applied and say that there are m such subsequences. The value returned would be

$$C(<e_1, e_2, \dots, e_n>, P_1, d_1)$$

$$\text{where } d_1 = C(<e_1, e_2, \dots, e_n>, P_2, d_2)$$

$$\text{where } d_2 = C(<e_1, e_2, \dots, e_n>, P_3, d_3)$$

...

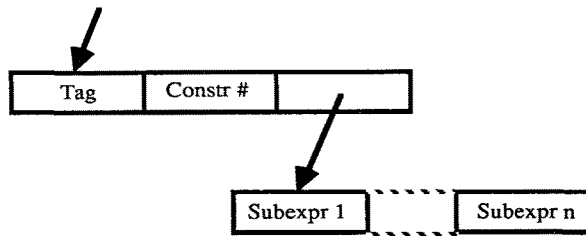
$$\text{where } d_k = d$$

This transformation has the biggest code expansion of them all, but it turns out that the kind of pattern that requires case 4 is rather rare in practice.

Code generation

As stated earlier the code generation proceeds in two steps, first G-code is generated, and from that machine code.

The code generation for a **case** expression very much resembles code generation for the case statement in ordinary languages, such as **Pascal**. The constructor number of the discrimination expression is used to select the match code. This code must then bind the variables in the (simple) pattern to the corresponding parts of the expression.



Constructor node layout. The tag indicates that it is a constructor node, the number is the constructor number, and the pointer points to a vector containing pointers to all the subexpressions.

The code generation for **case** expression with simple pattern can be defined in the same way as for the rest of the **LML** compiler, described in [John83] and [John84]. To simplify things the only code generation scheme that has to handle the **case** expression is the \bar{E} scheme, which generates code to evaluate an expression. In the places where the **case** expression should not be evaluated (as a subexpression to a constructor for instance) it is eliminated by a transformation. The transformation makes a function out of the **case** expression; the arguments to this function are the free variables in the **case** expression. The function can then be moved to the top level (all functions have to be on the top level for this compilation technique) and the original expression is replaced by a call of this function.

The code generated for evaluating a **case** expression (see below) first evaluates the discrimination expression, then performs a jump (**CASEJUMP**) to the code corresponding to the constructor number. This code first pushes all the subexpressions (**SPLIT**) of the discrimination expression and then evaluates the expression part of the case entry ($\bar{E} \ [e]$). All the variables of the pattern are available on the stack during this evaluation. After evaluating the new value, now on top of the stack, is moved to the right place (**MOVE**) and the discrimination expression and the subexpressions are popped (**POP**).

The code for the pattern which is a variable simply evaluates the code part (the value of the default variable can be found on the top of the stack).

The code for **default** pops all the unneeded things and jumps to the code for the surrounding default entry.

A few instructions are introduced to handle the pattern matching:

CASEJUMP which assumes that the top element of the stack is a constructor (as described above). As arguments it has a list of number, label pairs and a default label. It takes the constructor number and jumps to the corresponding label if it is present in the list, or else jumps to the default label. The actual machine code for this can be different depending how "dense" the case is. Normal compiling techniques can be used to produce good machine code.

SPLIT Assumes that the top element on the stack is a pointer to a constructor. It pushes all the elements of the vector part of the constructor node.

The *E* scheme below is not exactly the same as in [John84], it has been extended with the parameters *j* and *s*, which are the label to which a jump should be made and the amount to pop if there is a default.

```

E [case e in C1(v11,...v1n1):e1 || ... || v:en end] r m j s =
  E [e] r m j s; CASEJUMP (c1,l1) (c2,l2) ... ln;
    LABEL l1; SPLIT n1; E [e1] r[d1] (m+n1) l n1; MOVE n1+1; POP n1+1; JMP l
  ...
  LABEL ln; E [en] r[v=n] (n+1) j s; POP 1; JMP l
  LABEL l;

```

where d₁=[v₁₁=n, ... v_{1n₁}=n+n₁-1], ...
 c_k is the ordinal number of C_k.
 n_k is the number of subexpressions C_k.

```

E [default] r n j s =
  POP s; JMP j

```

A simple example:

```

case e in
  cons(a, b) : b
|| nil() : nil()
end

```

Assuming that *e* is on top of the stack, and the constructor numbers *nil*=0 and *cons*=1, the code for the previous example would be:

```

PUSH 0; EVAL; CASEJUMP (0,L1) (1,L2) L3;
  LABEL L1; SPLIT 2; PUSH 1; MOVE 3; POP 3; JMP L4;
  LABEL L2; SPLIT 0; MKNIL; MOVE 1; POP 1; JMP L4;
  LABEL L4;

```

From the G-code machine code can be generated quite easily. The CASEJUMP instruction can be compiled into a jump table (or compare and jumps if it is not dense). The SPLIT instruction turns into move instructions, moving the contents of the cell to the stack.

Acknowledgements

This work was supported by the Swedish Board for Technical Development (STU). The LML compiler has been developed in close cooperation with Thomas Johnsson. The Programming Methodology Group has provided a stimulating environment, and its members have given helpful comments. A special thanks to Thomas Johnsson, John Hughes, and Phil Wadler for reading and commenting an earlier version of this paper.

References

- [Augu84] L. Augustsson, "A Compiler for **Lazy ML**" in *Proceedings of the 1984 LISP and Functional Programming Conference*, Austin, Texas, (August 1984).
- [Burs80] R. M. Burstall, D. B. MacQueen, and D. T. Sanella, "Hope: An Experimental Applicative Language", pp. 136-143 in *Proceedings of the 1980 LISP Conference*, Stanford, California (August 1980).
- [Card84] L. Cardelli, "Compiling a Functional Language" in *Proceedings of the 1984 LISP and Functional Programming Conference*, Austin, Texas, (August 1984).
- [Frie76] D.P. Friedman and D.S. Wise, "Cons should not evaluate its arguments", pp. 257-284 in *Automata, Languages and Programming*, Edinburgh University Press, 1976
- [Hend76] P. Henderson and J.H. Morris, "A Lazy Evaluator", pp. 95-103 in *Principles of Programming Languages*, Atlanta, Georgia, (January 1976)
- [John83] T. Johnsson, "The G-machine: An abstract machine for graph reduction.", pp. 1-19 in *Proceedings of the Declarative programming workshop*, University College London, April 1983.
- [John84] T. Johnsson, "Efficient Compilation of Lazy Evaluation", in *Proceedings of the 1984 Symposium on Compiler Construction*, (Montreal 1984)
- [Miln78] R. Milner, "A theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, Vol. 17 no. 3 pp. 348-375 (1978)
- [Miln84] R. Milner, "Standard ML Proposal", *Polymorphism: The ML/LCF/Hope Newsletter*, Vol. 1 no. 3 (January 1984).
- [Turn76] D. A. Turner, *SASL language manual*, (1976)
- [Turn82] D. A. Turner, "Recursion Equations as a Programming Language", pp 1-10 in *Functional Programming and its applications*, Cambridge Press 1982.

Appendix A, LML code for the transformation

```

sort      -- sorts a list given a less than relation
group     -- groups a list into a list of lists given an equal relation
newvar    -- returns a new variable each time (not a real LML function)
subst     -- substitutes an expression for a variable
take      -- takes a predicate and a list and returns a pair: the longest prefix
           -- for which the predicate holds and the rest of the list
reduce    -- list reduction (also called fold or itlist)

```

```

choplist f [] = []
choplist f l = let (a,b)=f l in a . choplist f b

```

```

type rec Expr =      Ecase(Expr, list(Expr#expr)) +
                    Econstr(int, list(Expr)) +
                    Evar(id) + Edefault + Efail

```

the same type is used for both patterns and expressions.

```

let trans (Ecase(e,pes)) =
let partition p l = filter (not o p) l
and gsort lt l = group (\x.\y.not (lt x y) & not (lt y x)) o sort lt
and cfst (Econstr($,$).$,$) = true
|| cfst $ = false
in
let splitup = choplist (\l.let (cs,rs) = take cfst l in
                        let (ds,ns) = take (not o cfst) rs in
                        (cs@ds, ns))
in
let rec C [] [([], e)] def = e
-- Cases 2 and 3 are handled as instances of case 4 here
|| C (e.es) pses def = reduce C1 def (splitup pses)
    where C1 pses d =
        let (cs, ds) = take cfst pses
        in let css = gsort ltc cs
            where ltc (Econstr(n1,$).$,$)
                    (Econstr(n2,$).$,$) = n1 < n2
        in Ecase(e, map fc css @ [fd ds])
    where fd [] = newvar, d
    || fd a =
        let v = newvar
        in (v, C es (map (\(p.ps,e).(ps, subst v p e)) a) d)
    and fc (Econstr(n,nps).$,$).$ as a =
        let vs = map (\x.newvar) nps
        in (Econstr(n,s),
C (vs@es) (map (\(Econstr($, nps).ps, e).(nps@ps, e) a Edefault)
in
C [e] (map (\(p,e).([p],e)) pes) Efail

```