Translation of Pattern Matching and Other Context-Free Normalisation in Haskell

Brian Boutel
Research Report YALEU/DCS/TR-886
September 1991

This work is supported by grant NSF-CCR-9104987

Abstract

This document discusses some requirements for the "cfn" stage of the Yale Haskell compiler. This stage translates Haskell into a simple subset of itself, with the intention of simplifying the remaining stages of compilation. The most important task of the cfn processor is the translation of definitions which use patterns to select cases, and this is described in detail.

Translation of Pattern Matching and Other Context-Free Normalisation in Haskell

Brian Boutel

Yale University
Department of Computer Science
P.O. Box 2158 Yale Station
New Haven CT 06520
boutel@cs.yale.edu

December 18, 1991

1 Introduction

This document discusses some requirements for the "cfn" stage of the Yale Haskell compiler. This stage translates Haskell into a simple subset of itself, with the intention of simplifying the remaining stages of compilation. There has been some debate about the proper place for this processing to be performed, and in previous versions of the compiler it has been performed at an early stage, before typechecking , and even before scoping of variables (known as the "alpha" phase). It is now planned to postpone cfn (which stands for "context-free normalisation") until later in the processing. This has some advantage in that the typechecker will be processing a program that closely resembles the original, so error reporting will be easier to understand. The assumption that because typechecking has already been done, and so necessarily "alpha" processing has also been done, so that variable names are unique, simplifies the cfn processing.

The requirement that cfn processing is a Haskell to Haskell translation is taken as fixed. There is, however, some reason for believing that some of the translation would benefit from being from Haskell to some other code, such as the code for an abstract stack machine. Some of the alternative translations given attempt to indicate this by building tuples and selecting from them, rather than creating separate definitions which provide for the

extraction of each variable from a pattern. This approach would be similar to the stack machine code if the tuples were exploded onto the stack. At other times, scope considerations complicate the problem of sharing similar code. Again, a lower level target language could simplify this by branching to the appropriate label after ensuring the stack was consistently set up in all possible paths to the code.

2 Assumptions

1. Variables names are unique. That is, variables have been scoped, and a unique name associated with each distinct variable.

2. Sufficient type information is available. This does not necessarily imply that type inference has been done, only that it is possible to list the constructors of a type and their arities, given one of them. Even so, type inference must be done before context free normalisation, since function bindings are turned into pattern bindings in the translation, and monomorphism properties are therefore changed.

3 Translation of Expressions

Lambda Expressions: The translation is invoked by treating the lambda expression as an equation, and extracting the required result.

`\ p1 ... pn -> e` translates to
`res` where
`newvar = res` is the translation of
`newvar p1 ... pn = e`

Case Expressions: The translation is invoked by treating the case expression as an equation, and extracting the required result. The result will be a simplified case expression possibly qualified with definitions generated during the conversion.

`case e of alt1 ... altm`
translates to
`case e' of newalt1 ... newaltm`
where `e'` is the translation of `e`

A dummy name `newvar1` is chosen, and each alt

```
p | g -> e1 ...  where d1 ...
```
is rewritten as the equation
```
newvar1 p | g1 = e1 ...  where d1 ...
```
and the resulting set of equations is translated. The result is of the form
```
newvar1 = \ newvar2 ->
let { auxdefs } in case newvar2 of newalts
```
which is folded to give the final translation.

The resulting case expression may be optimised. This is discussed separately.

Arithmetic Sequences: The translation follows that given in the Report [1].

List Comprehensions: The translation given in the Report is reasonable, but it may be possible to do better. The problem with the translation given is the repeated processing of lists, e.g the initial translation for
```
 [ e | p1 <- l1 , p2 <- l2 ]
```
is, together with a definition of `ok`
```
concat (map (\p1 -> [ e | p2 <- l2]) (filter ok l1))
```
and the number of cons operations generated by `filter`, `map`, and `concat` is unnecessarily large.

It is probably better to define new higher-order functions to combine these operations. The following is suggested.

Define
```
comperp f p [] = []
comperp f p (x:xs) = let { rest = comperp f p xs } in
 if p x then f x ++ rest else rest
```
and
```
comperv f [] = [] comperv f (x:xs) = f x ++ comperp f xs
```
then
```
concat [e | v <- l]
```
where v is a variable, can be rewritten as
```
comperv (\v -> e) l
```
and
```
 concat [e | \p <- l]
```
for a pattern p as

```
let {ok p = True; ok _ = False} in comperp (\p -> e) ok l
```

Where no concat is generated, the form
`map (\p -> e) (filter ok l)`
can be rewritten as
`mapfil (\p -> e) ok l`
or
`map (\v -> e) l`
if the pattern is a variable, where

```
mapfil f p [] = []
mapfil f p (x:xs) = let { rest = mapfil f p xs} in
if p x then f x :  rest else rest
```

Other expression types are translated by simply translating their components.

4 Translation of Declarations

Only equations require any real work. These occur in Class and Instance declarations, and directly in Modules. They also occur within expressions, and the translation may be invoked from expression translation.

Syntactically, equations can be divided into pattern bindings and function bindings, and these are described separately.

4.1 Function Bindings

The translation of a function binding is to a binding of the function name to a lambda definition with new bound variables replacing the argument patterns. The right-hand-side is an expression, the structure of which is the main topic of this section.

```
f p11...p1n | g11 = e11 | ...  | g1k = e1k where { decls1}
 ...
f pm1...pmn ...
```

translates to

```
f = \ v1 ...  vn -> exp
```

Here the vs are new variables and the expression exp is formed as described below. The general form of the resulting code is a decision tree which takes the function arguments and selects the appropriate function body.

The standard procedure is described by Wadler in Peyton Jones' book [2]. It covers constructor patterns and variables, but does not describe the processing of literals, $n + k$ patterns, or irrefutable (~p) patterns. An alternative procedure is presented and discussed here, which has some performance advantages, but which, unless some redundent evaluation is done, is less strict than Wadler's method, which conforms to the semantics given in the Haskell Report.

4.1.1 Irrefutable Patterns

These are not described by Wadler, but the following transformation produces an equivalent program without them, to which either procedure may be applied.

The first stage of translation is to replace all irrefutable patterns defined by ~p by new variables and defer the pattern match to the deepest level of the expression.

```
...  ~p...  -> e translates to
...  v...  -> let
 x = (\p ->x)v
 ...
 y = (\p ->y)v
 in e
```

where `x...y` are all the variables in `p`. The alternative algorithm actually associates a value `(Twiddle p v)` with the expression `e` for later application of the translation.

4.1.2 Selecting the Argument to Match On

Wadler's method is to select the leftmost argument that is not a variable in all equations of the current group. This produces non-optimal code, in that tests may often be made more than once.

This method can be summarised as follows, (ignoring the processing of numeric constant and $n + k$ patterns):
Look at the first argument position and

- If it has a variable in all equations, drop it and continue with the next argument position.
- If it has a constructor pattern in all equations, generate a case expression dispatching on the constructor, assign each equation to a group

associated with the appropriate constructor, preserving the order of equations within each group, and process each group separately, dropping the used argument position, but prefixing the remaining arguments with the argument patterns of the constructor just matched.

- If it contains a mixture of constructor and variable patterns, partition the equations into groups that are all variable, or all constructor, maintaining the order of equations. Then process each group in turn, going to the next, or to error code in the case of the last group, when matching fails.

The problem with this method is that, in the code generated, the result of a test made in the course of checking one group will not be remembered in checking a later group if the earlier group fails to match, and so tests may be repeated unnecessarily.

The alternative algorithm attempts to achieve better performance by remembering the results of past tests, in order to generate code which will not repeat a test. It modifies the standard algorithm by not distinguishing between the last two cases, that is, it dispatches on the constructor even in the case where some of the equations have variables. Such equations can match any constructor, so they appear in every group. The order of equations in each group must be preserved.

The result of this is that a test which dispatches on the constructor of the actual argument partitions the equations into groups which are still possible matches given the result of the test, and further tests will be specialised for each group. This constrasts with the standard algorithm, in which the partitioning on a mixed argument position produces groups which have to be tested sequentially.

Selection of the argument position to dispatch on in this method is also different from the standard algorithm, in that the leftmost position that is not a variable in the first equation of the group is chosen. The strictness properties of this are at least as non-strict as the standard algorithm, but may depart from the semantics of Haskell in being too lenient, unless additional steps are taken.

In both algorithms, the following statement is true: "The argument position tested is always the leftmost untested position which does not correspond to a variable pattern in the first equation that has not already been rejected." The difference between the algorithms is that the new algorithm rejects some equations earlier than does the standard algorithm, and so may make fewer tests. The sequence of tests made by the new algorithm

is a sub-sequence of the tests made by the standard algorithm. The effect on strictness is that the new algorithm may avoid bottom-producing evaluations made by the standard algorithm, but will never diverge when the standard algorithm does not.

Unfortunately, the required semantics of Haskell are implemented by the standard algorithm, and to make the new algorithm conform, some additional argument evaluation may be necessary which does not contribute to the decision process of pattern-matching. Fortunately, however, these evaluations can be deferred until matching is otherwise complete, that is, the new algorithm can be used, noting for each outcome the additional computation required to be sufficiently strict, and performing this after matching. A compiler may choose, perhaps optionally, to ignore this extra evaluation.

As an example of this, consider the definition

```
f x [] [] = e1
f [] (a:b) c = e2
f x y z = e3
```

and the application `f bottom [] (p:q)`.
The standard algorithm will, in effect, do a naive first-to-last, left-to-right sequential test, because it partitions the equations into three single equation groups on the first argument. The actual sequence of tests will be

1. Match the second argument against `[]`, which succeeds
2. Match the third argument against `[]`, which fails, eliminating the first equation group
3. Begin testing the second group (the second equation) by matching the first argument against `[]`, and this diverges

The new algorithm proceeds differently.

1. Dispatch on the second argument. The `[]` value eliminates the second equation, leaving a possible group consisting of the first and third
2. Dispatch on the third argument. The `p:q` value eliminates the first equation, leaving only the third, which has no untested refutable patterns and is therefore matched

Here no test has been done on the first argument, because the only equation for which such a test is significant is eliminated before it can be made. To conform to Haskell semantics, an otherwise redundant test on the first

argument must be made. As noted above, this test can be made after matching is completed without affecting the strictness requirement, although forcing the evaluation implied by the test cannot readily be defined in Haskell code.

For Wadler's algorithm, no complications occur, as tests are performed in a sequence dictated by the desired strictness properties. The following sections describe the alternative algorithm.

4.1.3 Code Generation for the Dispatch - Constructor Patterns

The previous discussion has implicitly restricted itself to argument types which are algebraic data types with all patterns taking the form of variables or constructor plus component patterns. The introduction of constant or $n + k$ patterns is discussed in the next sections. In this section, the part of the algorithm to be used for argument positions where no constant or $n + k$ patterns occur is described.

A case expression is constructed, dispatching on the new variable name associated with the argument position, with an alternative for each constructor in the type of the argument.

Each alternative has

- a pattern consisting of a constructor and k new variables, where k is the arity of the constructor
- an expression containing code derived from further matching on the group of equations selected by the constructor

The equations in the group are those with the same constructor or an irrefutable pattern in the argument position.

There are opportunities for optimisation here, as some special cases of case expressions naturally translate to simpler code. This can be done once, for all case expressions, in the function that actually generates the concrete case expression code. A particular instance of this should be mentioned here, to prevent readers from worrying about it. This arises when the pattern is a tuple or other single constructor type. The code generated without optimisation for matching the pattern

`(p1,p2,...,pn)`

in a position named as v will be

`case v of(v1,v2,...,vn) -> case v1 of ...`

This is entirely satisfactory, and can be translated later to

```
let v1 = select ...  v...vn = ...in case v1 of ...
```

4.1.4 Code Generation for the Dispatch - Literals and $n+k$ Patterns

Should the selected position have a literal value, numeric constant, or $n+k$ pattern in any equation, the position is more complicated. The generated code will include a conditional testing for equality of the argument to the constant pattern or an ordering test for a $n+k$ pattern. These tests use the overloaded operations as defined for the type of the argument, and there is no guarantee that these operations obey the normal axioms. For example, if `0` and `1` occur as patterns, the matching tests to be generated for an argument `v` are (ignoring polymorphism) `equalT v fromInteger 0` and `equalT v fromInteger 1` where `equalT` is the equality function for the type T of the argument. There is no guarantee that these cannot both succeed, so assumptions used in the previous section to enable the elimination of equations which were known from information already obtained not to match are no longer valid, and opportunities to improve generated code quality are lost.

Another factor complicating code generation, is that equality or ordering comparisons may not require full evaluation of the argument. This is also true in the case of constructor patterns, but there only one test is required to partition the equations into candidate groups. In the present case, more than one test is often required. If the first test guaranteed the full evaluation of the argument, subsequent tests could be carried out immediately without violating the required strictness properties, but if not, it may be necessary to look at other argument positions first. Consider

```
f x 0 [] = e1
f [] 1 y = e2
f x y z = e3
```

If the equality test for 1 requires evaluation of part of the argument value not required for the equality test for 0, and that evaluation diverges, the divergence may have been avoided by matching the first argument against `[]` and failing. The semantics of Haskell require the less strict behaviour in these circumstances.

Another consequence of incomplete evaluation in comparisions is that to ensure correct strictness, the tests must be made in equation order, again possibly preventing the generation of improved code.

There are special cases where more is known about the behaviour of equality, ordering and conversion (e.g. fromInteger) operations, and this knowledge can be used to avoid some of the difficulties.

Strings A string literal pattern is equivalent to a list of characters. An argument of string type can have literal or constructor patterns, but no $n + k$ patterns. There is a choice of either treating the literals as constant lists, and so treating the argument as a list type and generating code which begins with a case expression selecting Nil or Cons, or using equality tests on the literals. Pragmatically, the latter course may be preferable, so that advantage can be taken of any optimised storage and comparisons for strings that the implementation may provide.

For strings, the equality function is the standard equality for lists of characters, and cannot be overridden, thus ensuring that static analysis can be used to eliminate equations with different literals if the equality test succeeds, and with the same literal if it fails.

Equality testing does not necessarily fully evaluate arguments, so literals must be tested in equation order, and allowance made for untested refutable patterns to the left of the literal.

A viable alternative is to treat strings as a list type, so that the simple case of constructor patterns without literals or $n + k$ patterns can be applied.

Character This type can have only constant and variable patterns, and an equality test fully evaluates its arguments. Equality is well-behaved in that two constant patterns representing different characters will not be equal. Code generation should therefore first generate an equality test between the argument and the constant in the first equation of the remaining group. (There will be a constant pattern in the first equation, since, if there were not, the position would not be matched on at this stage.) The test divides the equation into two sub-groups, those for which the test may return true are the first equation, any others with the same constant, and those with variables. The second group contains all equations with different constants and those with variables. These two groups can then be processed independently by the same method.

Should the character literals be closely spaced in the enumeration of characters, it might be more efficient to generate a range test, with a

case expression dispatching on the character value if it is in range, an a subset of the above code if not.

Numeric Types These are the most difficult to handle. They will be described in groups according to the kinds of problems that arise. Where type inference or explicit type signatures narrow the range of possible types for an argument position, the more specialised code can be used. Otherwise, general and generally inefficient code must be generated.

Int and Integer These types have known equality and ordering properties that can be taken advantage of at compile time. Furthermore, testing requires full evaluation.

There are no constructor patterns for these types. In the absence of $n + k$ patterns, they may be processed as for type `Char`. If the type is known, the constant values can be compiled directly to the appropriate value, otherwise code must be generated to convert the `Integer` constant to the appropriate type before it is tested.

If there are any $n + k$ patterns in the argument position, ordering tests must be generated. In addition, further code is generated later to provide the correct binding for the variable in the pattern. This is described below. Essentially, a new variable `v` is associated with the pattern, and the generated test is
`if v>=k then ...`,
with the then part containing code to bind `n` to `v-k`.

A problem introduced by $n + k$ patterns is that the ordering test does not normally divide the equations into two (possibly overlapping) groups, those that match and those that do not. There is usually a third group contining those equation that may match. For example, testing the pattern `n+2` creates a matching group excluding an equation with a constant `1`, including an equation with a pattern `n+1`, but leaving uncertain equations with constant patterns `3`, or with `n+3`.

For these types, since it is known that testing requires full evaluation, the changing the order of testing patterns in the various equations does not change the strictness properties. A possible strategy is then first to perform the equality tests in any order, and then the ordering tests in the order of largest k first. The advantage of doing this is that each test divides the equations

into two groups, the first, passing, group requiring no further testing on the argument position, and the second group possibly requiring further testing on the same position, but excluding the equation containing the pattern on which the test was based, thus simplifying the task of ensuring that the same test is not repeated.

The justification of this strategy is as follows.

An equality test of $v = c$ divides the equations into a group that matches if the test succeeds, containing equations with the same constant c, equations with $n + k$ patterns with $k \leq c$, and equations with variable patterns, and a group which *may* succeed if the test fails, containing equations with different constants, equations with $n + k$ patterns with $k > c$, and equations with variable patterns. For the first group, further testing passes to the next argument, while for the second group, the same argument may require further testing, but this group excludes equations with the same constant c, so this test will not be repeated.

A test of $v \geq k$, arising from an $n + k$ pattern, would divide the equations into three groups: those with $n + k'$ patterns with $k' \leq k$ which match if the test succeeds, those with $n+k'$ patterns with $k' > k$ or constant patterns c with $c \geq k$, which may match if the test succeeds, but do not match if it fails, and those with other constant patterns, which do not match if the test succeeds, but may match if it fails. In addition, equations with variable patterns belong to the first and third groups. Code generation from this is messy, and it is desirable to eliminate the second group. This is achieved by the suggested strategy.

Float or Double These types cannot have $n+k$ patterns, and so are a simple special case of the above.

Ratio Although this is a constructed type, the constructor is not exported from the Prelude, and so cannot appear in patterns. Neither can there be $n + k$ patterns for the type. What remains is constant patterns and variables, which can be treated as for Characters. For compiling the Prelude, constructor patterns are also needed, so a full implementation will instead treat Ratios in the same way as Complex.

Complex This types has constructor patterns, literals for which the mapping into values of the type is well-defined, but no $n + k$

patterns. In Haskell, both Ratio and Complex types have derived equality, based on structure, so equality tests for literals can be replaced by matching of constructor patterns, and matching for these types can be simple constructor pattern matching.

Arbitrary Numeric Type When insufficient information is available at compile time to determine that the argument type is one of the above types, the worst-case assumptions must be made. There may be locally defined or imported algebraic types with numeric instances, for which constant, $n+k$ and constructor patterns are all possible. Equality and order testing may not fully evaluate arguments. The user-defined fromInteger or fromRational functions and equality and ordering functions may be such that the normal laws are violated, so that, for example,
`fromInteger 1 == fromInteger 2`
has the value `True`, and
`fromInteger 1 < fromInteger 2`
has the value `False`.

With these possibilities, the appropriate strategy is to consider the first equation separately. If it has a constant or $n+k$ pattern, generate the appropriate equality or ordering test. In the then-part of the conditional, evaluated if the pattern matches, generate code to continue testing the first equation, but go on to test the group consisting of all the remaining equations should the first subsequentially fail, or match all argument patterns, but fail all guards. The else part should contain the same code for testing all remaining equations.

If the first equation has a constructor pattern, and any subsequent equation has a constant or $n + k$ pattern, a conditional should be generated to test the value of the top-level constructor. This is not available in standard Haskell, but a special *ifConstructor* function may be provided to implement it. The rest of the code is then as for constant or $n + k$ patterns. If no remaining equations contain constant or $n + k$ patterns, there is a choice between generating the special conditional as above, or generating a case expression dispatching on the constructors, as for simple constructor pattern argument types. If the case expression is chosen, later optimisation may well replace it with a series of conditionals if this is deemed more efficient, and so this

is the simplest strategy.

It is important to note that splitting a group of equations into the first and the rest, and continuing with the first until it fails, although reminiscent of Wadler's method, does not prevent application of the alternative algorithm in processing the rest of the equations.

Some improvement can be made to the generated code if it is known that the function cannot be used at any user-defined numeric type. This requires that it is not exported and no such types are defined in or imported into the current module. This ensures that no violation of the normal laws of arithmetic, equality and ordering is possible. In this case, static analysis can be used to exclude some equations from the groups carried along with the first equation in the then part of the conditional, and in the else part. This is similar to that described under Integer types above, i.e. for an equality test for a constant c excluding equations with a different constant or with an $n+k$ pattern with $k > c$ from the then part, and equations with the same constant c from the else part, and for an ordering test for a value $v \geq k$ excluding equations with constant $c < k$ from the then part and constant $c \geq k$ from the else part.

A practical compiler may choose to make these improvements even in the presence of user-defined numeric types, on the grounds that efficiency should not be compromised in the vast majority of cases by the remote possibility that someone has defined a type with a numeric instance, and given it nonsensical behaviour. On the other hand, the losses resulting from overloading make make any such savings immaterial.

4.1.5 Choosing the Next Argument

At any point in the translation there will be a list of argument positions not yet tested. Initially this will contain all arguments, and the number of arguments will be the same for each equation. There will also be a corresponding list of new variable names, one for each argument position. Initially this will contain variable names of the arguments of the top level lambda definition. The value of an item in the list of argument positions is the vector of the patterns in the equations of the group occurring at that position.

Conceptually, there is a pointer to the current argument position and corresponding variable name. At any time, argument positions before the pointer will not have been tested, but have been passed over. These will have irrefutable patterns (which will have been replaced by variables if they were originally ~p patterns) in the first equation of the group.

Choosing an argument to test involves advancing the pointer over irrefutable patterns in the first equation, and then generating code for the first refutable pattern. This will lead to the formation of new sub-groups of equations, which are then processed separately.

When a case expression is generated dispatching on the constructors of the type a new sub-group is formed for each constructor. Further testing involves matching the components of the matched constructor in the actual argument against the corresponding components of the constructor in the patterns. For each equation, these components are inserted into the remaining pattern list in place of the original pattern. Where the original pattern was a variable (standing for any irrefutable pattern) a list of variable names is inserted instead, the same list for each such occurence. In the associated list of variable names, the name corresponding to the whole pattern is deleted, and the same list of variable names inserted. This contains the same variable names as were added to the constructor to form the left side of the current alternative in the case expression generated for the argument position test.

When a conditional is generated, two or possibly three subgroups are formed. The first sub-group, the candidate equations if the test succeeds, in the case of an equality or ordering test or an "ifConstructor" test of a nullary constructor, require no further test on the same position. The argument position matched on is dropped from the list of remaining patterns for each equation in the group, and the corresponding variable name from the associated list of variable names. For an "ifConstructor" test, if the arity of the constructor is non-zero, the list updating is as for case expressions

The other sub-groups will require the same argument position to be tested again, and no change is made to the lists.

The choice of next position to match requires a return to the beginning of the remaining pattern list. The position just matched has in general not fully evaluated the argument, and there may be positions to its left that have not yet been full evaluated for which refutable patterns are now exposed.

4.1.6 Termination

In the selection code generation described above, before conditional or case expression code is executed, there is a group of equations which are still possible candidates for evaluation of their right-hand-sides. After execution of the conditional or case expression code, one of a number of possibly overlapping sub-groups of equations becomes the new candidate group.

The generated code as described above always has one or both of the following properties:

- The new candidate group is a proper subset of the old group,
- The argument position used for selection is not used again.

To see this, first observe that the only cases where the same argument position is used again are the "reserve" group in the then-part of a conditional, or in the else-part. In all cases, the condition tested is such that the pattern giving rise to the test satisfies it, and so is excluded from these candidate groups.

From this it is clear that in any path throught the selection code, a point will be reached where either all the arguments have been used, or the candidate group is empty. In either case, no further selection code is generated.

4.1.7 Selection and Translation of Right-Hand-Sides

In the ideal case, the selection process will leave a single candidate equation after testing all arguments. In this case, the right-hand-side of that equation will be evaluated. Should the candidate group be empty, error code is generated.

Where the candidate group still has more than one equation after testing all arguments, the semantics specify that the first equation is selected. It is possible that this equation has guards, and that all the guards may fail, in which case the remaining equations are treated as a subgroup and tested, returning the argument position pointer to the beginning of the list before searching for the first argument to match. A similar situation arises when a conditional test creates a "reserve" group.

Translation of a guarded right-hand-side is to a conditional expression with the guard as the if-part, the right-hand-side as the then-part and the translation of the remaining guarded right-hand-sides as the else-part. If the guard is the constructor `True`, or an identifier known to be bound to `True`,

or absent, no conditional is necessary and the code generated is simply the code for the right-hand-side. In this case, any remaining guarded expressions in the equation can be ignored.

The final else-part, should one be required, is the translation of the remaining equations in the group. This translation in general requires the testing of arguments passed over in previous code because they were irrefutable in the first equation.

During the processing of pattern matching, some information is collected which results in further transformation of the expression. This occurs in the translation of ~p patterns and as-patterns, and from the loss of some original variable names from patterns, which may occur in function bodies.

- For ~p patterns, the pattern is not to be matched until a variable mentioned in the pattern is required in evaluation of the body, when the whole pattern is matched. The translation is shown above. Note that the result contains lambda definitions, which need to be processed by the expression translator.

- For as-patterns, `v@p`, a new name `v'` is introduced in the translation for the value matching p, while the body may still refer to v.

- Variables in constructor patterns are also replaced by new variables.

- For $n + k$ patterns,
  `f ... (n+k)... = e`
  becomes `f ... v'... = (\ n ->e) (v' - k)`

Also, as function bodies may be reached from several paths in the pattern matching decision tree, to avoid code replication, they are given names, the names referred to in the decision tree and definitions of the form
`name = body` attached to the tree code. This moves the bodies out of scope of the variable names defined in the case alternatives. There are thus two problems: variables defined in patterns and occurring in the function bodies have been renamed, and possibly renamed differently in different parts of the decision tree code; and the bodies have been lifted out of the scope of the definitions.

The solution to this is to complete the lifting by turning the bodies into functions of the original pattern defined variables, and at each occurence of a body at a leaf of the decision tree, applying the function to arguments which refer to the new pattern variables. This is an unfortunate consequence of generating pattern matching code in Haskell. A direct generation of code for

an abstract stack machine such as Cardelli's Functional Abstract Machine can avoid these problems.

For efficiency, only those pattern-defined variables ocurring in the body need be defined. The information available, and the generated code, is

- For an irrefutable pattern ~p, a value `Twiddle p v`, giving the new name `v` associated with the pattern, which will itself contain information about further variable names. For each variable `x` occurring in both ~p and the body, a formal parameter `x` is abstracted from the body, and the argument value `(\ p -> x) v` is added, where `v` will vary from occurence to occurence of the body. To maintain uniqueness of variable names, a new variable, `x'` is actually used instead of `x` as the formal parameter, and `x` renamed to `x'` in the body.

- For as-patterns and renamed variables, a value `As v v1` where `v` is the name replaced by `v1`. If `v` occurs in the body, all occurrences are replaced by a new variable name `v'`,which is also abstracted out as a parameter. The actual parameter will be `v1`.

- For $n + k$ patterns, a value `Offset n k v` . If `n` occurs in the body, all occurrences are replaced by a new variable name `n'`,which is also abstracted out as a parameter. The actual parameter will be `v - k`.

Finally, the modified body expression is translated as an expression.

4.2 Pattern Bindings

The binding
`p | g1 = e1 ; ... ; | gn = en where d`
is first translated to
`p = let d' in if g1' then e1' else ... else error`
where the primes indicate that the values have been individually translated. If a guard is absent, is the constructor `True`, or is an identifier known to be bound to `True`, no conditional is generated and any remaining guarded expressions or error expressions discarded.

Translation of the resulting definition produces a set of declarations which may themsleves require translation. The process of translation is continued until no further translation is required. Translation is defined as follows

- The as-pattern usage `v@p = e` translates to

```
y = e
x1 = (\p'->x1') y
...
xn = (\p'->xn') y
v = (\p->y) y
```

where the `xi` are the variables in `p`, `p'` is `p` with all `xi` renamed to `xi'`. Equations for those `xi` and `v` which do not occur in `e` need not be generated.

An alternative translation is

```
y1 = e
y2 = (\p -> (x1,x2,...,xn,y1)) e
 ...  xi = select i y2 ...
 v = select (i+1) y2
```

The tradeoff between these two is that the first requires a full run-time pattern match for every pattern variable occurring in `e`, while the second requires only one match, and a simple selection from a tuple for each needed variable, but builds the tuple.

- With definitions as before, the irrefutable pattern usage

```
~p = e
```

should translate to

```
y = e; ...  xi = (\p' -> xi') y ...
```

or

```
y = (\p -> (x1,x2,...,xn)) e
 ...  xi = select i y ...
```

This is very similar to the as-pattern translation.

- With the same definitions, the constructor pattern usage

```
C p1 ...  pm = e
```

translates to

```
y1 = e
y2 = if ifConstructor C y1
 then
 (\p1...pm -> (x1,...,xn)) (select 1 y1)...(select m y1)
else error
 ...  xi = select i y2 ...
```

In the case of a tuple or other single-constructor type, the conditional

can be omitted, and the then-part alone, together with the selection definitions, provides the translation.

A useful optimisation can be applied here if the sub-patterns `pi` are all variables. The form
`(\ p1...pm -> (x1,...,xn)) (select 1 y1)...(select m y1)`
reduces to
`(\ x1...xm -> (x1,...,xm)) (select 1 y1)...(select m y1)`
which is
`((select 1 y1), ..., (select m y1))`
and this tuple is only built to be selected from. Instead, the form

```
y1 = e
y2 = if ifConstructor C y1
 then y1
 else error
 ...  xi = select i y2 ...
```

is equivalent. If the conditional is omitted as above, the definition of y2 can be dropped and y1 used instead.

The case expressions resulting from constructor patterns are candidates for optimisation.

- For Literal and $n + k$ patterns we note that all the above cases are applied only at the top level, and all subsequent processing of pattern bindings, as opposed to variable bindings, is in the context of function bindings. Consequently, we need consider literals and $n + k$ patterns at the top-level only.

  Definitions with a literal left hand side may be ignored. Indeed, in all of the above, a definition with no variables in the left hand side may be ignored and dropped form the program. Such definitions bind no variables and cannot contribute to any expression evaluation.

  A definition of the form `(n+k) = e` has the translation
  `let { v = e ; n = v - k } in if v >= k then v else error.`

5 Optimisation of Case Expressions

There are a number of occasions when the case expression generated in a translation may be better rendered by other concrete code.

One such time is when the case has only two alternatives. The obvious code to generate in this case is an "if then else" conditional. When the case

expression is a dispatch on the constructor of a value, the special construct `ifConstructor` must be used in the if part, even if the constructor is nullary. It is incorrect to test, for example `if v == C1 then ... else ...`, as the type of which `C1` is a constructor may have user-defined equality which differs from the structural matching normally associated with pattern matching and implied by `ifConstructor`.

If the alternatives bind variables, auxiliary definitions must be introduced. For example, the translation of

```
case v of { [] -> e1 ; (x:p) -> e2 }
```

is

```
if ifConstructor v [] then e1 else
let { x = select 1 v ; v1 = select 2 p} in case v1 of ...
```

If the expression has more than two alternatives, but all but one have the same outcome, for example error code, then a similar conditional can be generated. This will often arise from a pattern binding with a top level constructor pattern.

Sometimes the case expresion has only one alternative. Tuples are an obvious example of how this can come about, although any single-constructor type will also produce it. It differs from the previous case because there is no error option, and no test on the constructor need be done. Omitting the case expression loses the bindings of variables, and these must be replaced by local definitions. Typically,

`case v of { (v1,...,vn) -> e }`

becomes `let {v1 = select 1 v ; ... } in e`.

More generally, there is an efficiency tradeoff between a case expression and cascaded conditionals. This is something that should be left to a later stage of compilation where the characteristics of target machine code are known. There are some obvious examples where jump table is practicable, e.g. in a dispatch on constructor, or with a guarding range test when there are a number of small integer constants and the type is known to have a sensible equality definition, and these cases should obviously not be translated to conditionals.

References

[1] Paul Hudak, Simon Peyton Jones, *et. al.*, *Report on the Functional Pro-*

gramming Language Haskell, Version 1.1, Yale University, Department of Computer Science, August 1991.

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