SML/NJ Match Compiler Notes

William Aitken

1992

The match compiler is used to compile pattern matches into lambda code. Its input consists of a list of pairs of abstract syntax patterns and lambda expressions. It produces a lambda expression that denotes a function. This function behaves as if it matches a single argument value against each of the input patterns in turn until a match succeeds, makes the bindings required by this match, and executes the selected rule's lambda expression in the resulting environment.

The generated lambda expression is constructed to implement a decision tree of primitive tests. Various principles guide the production of this lambda expression. First, no memory access should be performed more than once. Second, no test should be perfomed more than once. Third, no unneccesary memory access should be performed. Fourth, memory accesses should be performed as soon as they can be shown necessary. Because case selection is implemented as a decision tree, code duplication may occur. The match compiler takes steps to try to minimize the effects of this duplication.

Patterns may are built up from constructors, variables, and wild cards using record construction, vector construction, layering, type constraints and constructor application. There are various kinds of constructor. First, there are *special constructors* like `1`, `1.0`, and `"str"`. They are the constants provided by Standard ML for the integer, real and string types. Second, there are *primitive constructors*. These are the value constructors provided by datatype and exception declarations (and also `ref`). Third, there are *concrete constructors*. These are constructors declared using the new template mechanism, and whose expansion is currently available. Last, there are *abstract constructors*. These are constructors specified by a template specifiaction whose representation is unknown. For example, in the body of the functor declared below, `Arg.abs` is abstract, `conc` is concrete, `prim` is primitive and `1` is special.

```
signature ARG =
  sig
    const abs : int
  end
```

```
functor(Arg : ARG) =
  struct
     datatype dt = prim of int
     const conc = prim(1)

     fun test Arg.abs = true
       | test _ = false
  end
```

Because different patterns may refer to the same component of the value being matched with different variables. It is convenient to adopt a uniform naming scheme for subcomponents of the value being matched. To this end we introduce *destructuring paths*. A destructuring path may be the root path, which when applied to a value returns the value. It may also be a record projection of a path, the vector projection of a path, the vector length of a path, or the constructor deconstruction of a path. When applied to a value, these paths apply their argument path to the value, and perform the appropriate destructuring operation on the resulting value. Finally, a path may be a tuple of paths. When applied to a value, a tuple path applies each of its argument paths to the value, and forms the tuple of the resulting values.

An *extended pattern* is just like an ordinary pattern except that it may contain an abstract constructor or an abstract constructor application *anywhere* that a variable may appear, and it may contain vector patterns wherever record patterns are legal. It may not contain concrete constructors or or–patterns. Note in particular that abstract constructors may appear in layering patterns.

1 Match Rule Preprocessing

The first step of match compilation involves preprocessing the match rules into a more amenable form. First, we ensure that the lambda expressions of the match rules are not duplicated. Second, we remove all or patterns. Third, we expand concrete constructors. These steps are described in detaill below.

For each match rule, a new temporary variable f is created, and a pair of objects returned. The first of these is a sequence of triples l. The first element of each of the triples is a sequence of pairs of destructor paths and extended patterns. The second is a sequence of destructor paths. And the third is the variable f. l represents the value destructuring and case testing effect of the match rule. The sequence of pairs is a “simplified” version of the pattern, the sequence of destructuring paths are the paths of the variables of the pattern. The second result object is a pair consisting of the variable f and a lambda term. This pair represents the variable binding and expression execution effect of the match rule. The lambda term performs these operations. f is used for communication between the two phases of pattern matching.

To execute a match rule on a value v — that is, to match the value against the match rule's input pattern, and to then execute the match rule's expression if the match is successful, and fail if it is not — we take each triple of l, in order. Each triple consists of a sequence m of pairs of destructor paths and extended patterns, a sequence of destructor paths a and the variable f. For each element of this sequence m, we apply the destructor path to the value to be matched v, and match the resulting value against the corresponding pattern. If all these matches succeed, we apply f. Otherwise, we continue with the next element of l. If we run out of elements of l before we find one where f is applied, the match fails. The value to which f is applied depends on a. If a is empty, we apply f to the 0-tuple. If a contains only one destructor path d, we apply the destructor path to v, and apply f to the resulting value. Otherwise, we apply each of the destructor paths of a to v, form a tuple of the resulting values in order, and apply f to this tuple.

The second component is a pair containing the variable f and a lambda expression. The lambda expression is constructed in such a manner that if f is bound to it, the applications of f described above are equivalent to executing the match rule's expression with the appropriate new bindings.

The production of this value requires several suboperations to be performed. We describe thm below.

1.1 Expression Preprocessing

The lambda expressions associated with patterns in the input match representation may be very large. For this reason, it is essential that they not be duplicated. Moreover, it is highly desirable that they not be examined, but rather treated as atomic, abstract objects. To avoid the duplication of these lambda expressions, we locally bind a function for each match rule. These functions bind the variables bound by the match rule's pattern, and execute its lambda expression. At the leaves of the decision tree, we generate calls to these functions rather than copies of the match rules' lambda expressions. This has the further advantage of obviating the need to rename variables in the lambda expressions to maintain the single binding property of lambda code. No attempt is made to limit the bindings of these functions to those variables actually used by the corresponding lambda expression, or to avoid generating a function for very simple lambda expressions since both these operations would require examination of the lambda expressions.

The function produced for a given match rule depends on the variables bound by the rule's pattern and on the rule's lambda expression. If the pattern binds no variables, the function is denoted by a lambda term that binds a new temporary variable, and has the rule's lambda expression as a body. Leaves corresponding to the rule consist of applications of this function, or rather, the variable to which it is bound, to the unit record. If the pattern binds a single variable v, the function is denoted by a lambda term that binds v, and has the rule's lambda expression as a body. Leaves corresponding to the rule consist of applications of this function to the value of v induced by the pattern match. If the pattern binds more than one variable, $v_0, \ldots, v_{n-1}$, the function is denoted by a lambda term that binds a new temporary variable x. The body of this term consists of a series of nested local bindings in which a variable v_i is bound to the ith record component of x enclosing the rule's lambda expression. The local bindings are implemented in the usual manner with an application term that has a lambda term as its function component. Leaves corresponding to the rule consist of applications of this function to a tuple containing, in order, the values bound by the pattern to $v_0, \ldots, v_{n-1}$.

For each match rule, this phase of the compilation process produces three objects: the lambda term produced for its right hand side, the variable name to which it is to be bound, and an ordered list of the variables bound by the rule's pattern.

1.2 Or Pattern Expansion

Next, or patterns are expanded according to the following rules.

$$\frac{}{v \Rightarrow v}$$

$$\frac{}{scon \Rightarrow scon}$$

$$\frac{}{\_ \Rightarrow \_}$$

$$\frac{}{con \Rightarrow con}$$

$$\frac{pat \Rightarrow pat_1, \ldots, pat_n}{pat : ty \Rightarrow pat_1 : ty, \ldots, pat_n : ty}$$

$$\frac{pat \Rightarrow pat_1, \ldots, pat_n}{p \texttt{ as } pat \Rightarrow p \texttt{ as } pat_1, \ldots, p \texttt{ as } pat_n}$$

$$\frac{pat \Rightarrow pat_1, \ldots, pat_n}{con\, pat \Rightarrow con\, pat_1, \ldots, con\, pat_n}$$

$$\frac{pat_1 \Rightarrow pat_{11}, \ldots, pat_{1n_1} \quad \cdots \quad pat_m \Rightarrow pat_{m1}, \ldots, pat_{mn_m}}{-pat_1, \ldots, pat_m{}'' \Rightarrow -pat_{11}, \ldots, pat_{m1}{}'', \ldots, -pat_{11}, \ldots, pat_{mn_m}{}'', \ldots, -pat_{1n_1}, \ldots, pat_{m1}{}'', \ldots, -pat_{1n_1}, \ldots, p}$$

$$\frac{pat_1 \Rightarrow pat_{11}, \ldots, pat_{1n_1} \quad \cdots \quad pat_m \Rightarrow pat_{m1}, \ldots, pat_{mn_m}}{\#[\, pat_1, \ldots, pat_m] \Rightarrow \#[\, pat_{11}, \ldots, pat_{m1}], \ldots, \#[\, pat_{11}, \ldots, pat_{mn_m}], \ldots, \#[\, pat_{1n_1}, \ldots, pat_{m1}], \ldots, \#[\, pat_1}$$

$$\frac{pat \Rightarrow pat_1, \ldots, pat_n \qquad pat' \Rightarrow pat'_1, \ldots, pat'_m}{pat \,—\, pat' \Rightarrow pat_1, \ldots, pat_n, pat'_1, \ldots, pat'_m}$$

This produces a list of patterns for each match rule's pattern.

1.3 Template expansion

Next, templates are expanded in each of the patterns produced by or expansion; that is, concrete constructors, and applications of concrete constructors are expanded as described in Section 2. For each pattern, this produces a replacement extended pattern, a list of constraints, and list of variable instantiations.

As described in Section 2, the expansion of a concrete constructor k yields a replacement extended pattern and a list of constraints. There are no variables to instantiate in k.

To expand a concrete constructor application $k\,p$, first expand the subpattern p to yield an extended pattern xp, a list of constraints c and and a list of variable instantiations ρ. Expand the template k with the argument extend pattern x. This yields an extended pattern xp', a list of constraints c' and a list of variable instantiations ρ'. The result of of expanding $k\,p$ is the extended pattern xp' together with the constraints of c and c' and the variable instantiations of ρ and ρ'. The expansion of k might fail. In this case, the pattern being expanded is unmatchable.

Special constructor, non-concrete constructor and wild card patterns expand to themselves. No constraints or variable instantiations are added.

An application of a non-concrete constructor $k\,p$ is expanded by expanding p to produce an extended pattern xp, a list of constraints and a list of variable instantiations. The replacement extended pattern is formed by applying k to xp. This is returned together with the constraints and variable instantiations arising from the expansion of p.

All other patterns are expanded by expanding their subpatterns, The resulting replacement extended patterns are recombined in the same manner as the patterns from which they arose were combined to produce the replacement extended patterns. The resulting constraints are appended together, as are the resulting variable instantiations to produce the returned constraints and variable instantiations respectively.

1.4 Path replacement and clean up

We can now compute the required destruction path for every variable that occurs in the replacement or constraint extended patterns returned by template expansion.

As a result of template expansion, some or all of the variables bound by each pattern may have been instantiated. Moreover, because the expansion is iterative, variables occuring in constraint trivial patterns and in variable instantiations may also have been instantiated. We repeatedly use the variable instantiation to replace instantiated variables of the pattern binding list and constraint trivial patterns, until no further instantiated variables remain. It is now the case that every variable occuring in the constraint trivial patterns or the binding list has been bound by one of the extended patterns returned by template expansion. We replace the constraint trivial patterns and each element of the pattern binding list with the appropriate destructor path: variables are replaced by their path, records are replaced by tuple paths that have the translations of their subcomponents as subpaths.

We build a pair consisting of of the root destructor path and the replacement extended pattern. We add this pair to the list of constraint pairs (with their trivial patterns converted to destructor paths). We form a triple consisting of this list, the list of paths arising from the pattern binding list, and the variable name to which the right hand side function for this match rule is to be bound.

1.5 Returning

We form a list of these triples, and pair it with the pair consisting of the function name and body. This is the result of preprocessing a single match rule. We perform these operations for each match rule.

2 Template Expansion

The template expander is responsible for eliminating concrete constructors. Informally, it is responsible for replacing patterns of the form $k\,p$ or k, in which k is a concrete constructor, and p is a pattern, with equivalent patterns that include no concrete constructors. Unfortunately, for various reasons, its operation is somewhat more complicated than this ideal. We highlight the additional complexities as they arise in the discussion below.

Suppose we wish to expand a pattern of the form $k\,p$ in which k is a concrete constructor declared by a declaration of this form:

$$\texttt{const}\ k\ t = e$$

and in which p contains no concrete constructors. Assume further, for now, that e contains no concrete constructors. Intuitively, we match p against t to obtain a substitution for the variables of e — recall that in a template declaration every variable occurring on the right hand side must also occur on the left hand side — apply this substitution to obtain the replacement pattern p'. Since neither p nor e contain concrete constructors, neither does p'. Unfortunately, various problems arise when we attempt to do this in practice.

First, the constructor's argument pattern pattern p may have less structure than its trivial pattern t. To make the discussion concrete, we introduce the following constructor definition.

```
const eg1 (a, b) = [a, 1, b]
```

Consider the pattern `eg1 _`. The argument of `eg1` in this pattern has less structure than the trivial pattern `(a, b)`. In situations like this one, the solution is simple: give the wildcard as much structure as is needed to make the match succeed. This is possible because wildcards match anything and bind nothing. In this case, `_` is replaced by `(_,_)`; and so, `a` and `b` are both instantiated to wildcards. The result pattern is `[_, 1, _]`.

Now, consider the pattern `eg1 x`. Once again, the argument pattern has too little structure. The solution used previously no longer works because the variable `x` must be bound to the entire argument of `eg1`. Instead, we replace `x` with a trivial pattern with the same structure as the trivial pattern, but with new temporary variables instead of the trivial pattern's. We arrange that this trivial pattern (which is also an expression) be used instead of `x` in the scope of the pattern. The match can now proceed, and the variables of the trivial pattern will become instantiated with the newly allocated temporary variables. For example, in this case, we replace `x` with `($1, $2)`. `a` is instantiated with `$1`, `b` is instantaiated with `$2`, and the resulting pattern is `[$1, 1, $2]`. We must also arrange that the expression `($1, $2)` be used instead of `x` in the scope of the pattern. There is no need to update the rest of the pattern since `x` only appears once in the pattern. A similar problem arises because of layered patterns. For example, consider the pattern `eg1 (x as (`p_1`, `p_2`))`. We handle such examples by instantiating `x` with a sufficiently structured trivial pattern, and merging this pattern with the body pattern. In this case, this results in the argument pattern being replaced with the pattern `($1 as `p_1`, $2 as `p_2`)`. We must also arrange that `($1, $2)` be used instead of `x` in its scope. We discuss the process of merging two patterns in more detail later.

Now, consider the pattern `eg1 abs0`, in which `abs0` is a nullary abstract constructor. Here we must somehow arrange that the entire argument value of `eg1` be matched against the constructor `abs0`. Our solution is to treat `abs0` as if it were a variable, instantiate it to a trivial pattern with sufficient structure, and add a global constraint saying that this trivial pattern must match the abstract constructor. In this case, this means that the resulting pattern is `[$1, 1, $2]`. A constraint demanding that the expression `($1, $2)` must match `abs0` is added. Note that this constraint requires that a record match an abstract constructor. This is important, beacuse is ensures that the constraint doesn't introduce aliases for subcomponents of the value being matched. A similar situation arises for patterns like `eg1 (abs1 `p`)`, in which `abs1` is a unary abstract constructor. An analagous solution is used. In general constraints always specify that a non-variable trivial pattern match an abstract constructor, or the application of an abstract constructor.

Earlier, we assumed that the patttern expression of a template contains no concrete constructors. Of course, this is not *a priori* true. To make it true, we must expand the pattern expression when the template is declared. Unfortunately, in addition to replacing the pattern expression with a pattern free of concrete constructors, this may also specify that the variables bound by the pattern expression become more structured, and it may add constraints. The scope of the pattern expression of a template is the template's trivial pattern.

Fortunately, this can be changed, so the structuring of variables is no great difficulty. The addition of constraints means that in addition to a trivial pattern and a pattern expression, a template might bind a set of constraints. To specify the constraints, variables appear in the pattern expression that do not appear in the trivial pattern. These variables must be instantiated with new temporary variables whenever the template is instantiated to ensure that patterns remain linear even if they contain multiple instances of the same template. Otherwise, the addition of these constraints causes no difficulties: their variables are replaced, either by new temporaries, or by the appropriate binding resulting from the match between the constructor's trivial pattern and its argument. And they are then added to the constraints generated by the match, when a template is instantiated.

Another problem arises because the right sides of projection declarations may be arbitrary patterns. For example,

```
proj eg2 (a, b) = a as (1,b)
```

Naively substituting for `a` in the right side will result in an illegal pattern unless the value being substituted for `a` is itself a variable. Our solution here is to merge the result of substituting on `a` and the result of sustituting on `(1, b)`.

The merging of patterns, alluded to above is done as follows: A variable v is merged with a pattern p by layering to yield the pattern v `as` p. A wild card pattern is merged with a pattern p by ignoring the wildcard to produce p. Two record or vector patterns are merged pointwise. Two equal special constructor or primitive constructor patterns are merged by ignoring one of them. An attempt to merge two unequal special constructor or primitive constructor patterns means that the pattern is unmatchable. Merging two application patterns that apply the same constructor results in a pattern that applies this constructor to the result of merging their subpatterns. Attempting to merge two application patterns formed with different primitive constructors, or an application of a primitive constructor and a primitive constructor implies that the pattern is unmatchable. Merging a layered pattern v `as` p with a pattern q results in the pattern v `as` r, in which r is the pattern obtained by merging p and q. Finally, we must consider merging abstract constructors and their applications with patterns. This is most easily done if we allow them to appear as the layering in layered patterns. Then we can merge an abstract constructor k with a pattern p by forming the layered pattern k `as` p. And merging the abstract constructor application $k\,q$ with p yields $k\,q$ `as` p.

Only two complications arise because of this extension. First, patterns with abstract constructors (or their applications) as layerings may appear in the output of the template expander. Second, the match against trivial patterns must handle them. To match a pattern k `as` p against a non-variable trivial pattern t, treat the pattern as if it were an ordinary layered pattern v `as` p. This will yield an instantiation for the variables of t, and a replacement trivial pattern s for v. Add the constraint that s must match k. Layerings involving abstract constructor applications are treated similarly.

Expanding a nullary concrete constructor is similar, but simpler because there is no subpattern and trivial pattern. Simply replace any variables in the constant or it constraints with fresh temporaries (to ensure that patterns remain linear), replace the constructor with the replacement pattern, and add any constraints to the patterns constraint set.

In summary, template expansion takes a unary concrete constructor and an argumenet extended pattern and produces a replacement pattern, a list of constraints and a list of replacements for variables of the argument pattern. constant expansion takes a nullary concrete constructor and produces a replacement pattern and a list of constraints.

3 Pattern Transposition

Because we wish to avoid repeated tests, and because we wish to effectively utilize case terms in our lambda expression, it is convenient to gather all tests of a particular component of the value to be matched. This is done by building an *and-or tree*. Another way to look at this is to consider it as transposing the match. Instead of arranging the tests to be performed by the rule in which they occur, we arrange them by the portion of the matched value they test.

If our only concern were the compilation of ordinary Standard ML patterns, this would straightforward. We would include three kinds of nodes in our and-or trees. First, *leaf–nodes*. Second, *and–nodes*. These would consist of an ordered list of and–or trees. Third, *or–nodes*. These would consist of a list of triples. These triples would contain a guard, a list of selected rule numbers and an optional and–or tree. The guards might be integers, reals, strings or constructors, and all case selectors of a particular or–node would have the same type.

To produce an and–or tree from a sequence of match rules, we would first choose a match rule and generate a match rule for its pattern, and then merge in the patterns of all the other match rules, one at a time.

The generation of an and–or tree from the pattern p of rule n proceeds as follows. If p is a variable or wildcard pattern, generate a leaf–node. If p is a special constructor or nullary constructor pattern, generate an or–node with a single triple containing the constructor, a singleton list of n, and no subsidiary and–or tree. If p is a constructor application $k\,p'$ generate an or–node with a single triple containing the constructor, a singleton list of n, and the tree generated for p' as a subtree. If p is a record pattern with m sub-patterns, generate an and–node with m subtrees, one for each sub-pattern. For a constrained pattern $p' : \tau$ generate the and–or tree for p'. For a layered pattern p_1 as p_2, generate the and–or tree for p_2. or–nodes are designed in this way so that the guards correspond with the guards of case statements in the lambda expression language.

Merging the pattern p for rule n with an and–or tree t proceeds as follows. If t is a leaf–node then simply generate the and–or tree for rule p. If p is either a variable or wildcard pattern, simply return t. If p is either a layered or constrained pattern, merge the body pattern of p with t. If p is a special constructor or nullary constructor pattern, and t is an or–node that includes a triple with the constructor of p as its guard, and no subtree, then return the tree formed from t by adding n to the list of selected rules in this triple. If p is a special constructor or nullary constructor pattern, and t is an or–node that does not include a triple with the constructor of p as its guard, return the tree formed from t by adding a triple consisting of the constructor of p, the singleton list containing n, and no subtree. If p is the application of a constructor $k\,p'$, and t is an or–node that includes a triple guarded by k with a subtree t', return the tree formed from t by adding n to the list of selected rules in this triple, and merging p' with t'. If p is the application of a constructor $k\,p'$, and t is an or–node that does not include a triple guarded by k, return the tree formed from t by adding a triple consisting of the constructor of p, the singleton list containing n, and the and–or tree generated from p' as a subsidiary tree. If p is a record pattern with m sub-patterns and t is an and–node with m subtrees, return an and–node with subtrees formed by merging the corresponding components of p and t. The type system guarantees that these are the only possibilities.

For us, the situation is somewhat more complicated. First, we must handle vector patterns. Second, we must handle abstract constructors. Third, variable and layered patterns need to be treated somewhat more carefully.

Vector patterns are handled by extending the class of guards allowable in or patterns to include vector length guards, and allowing lists of subtrees in or–nodes rather than optional subtrees. Where there was no subtree before, we place a nil list. Where there was a subtree before, we place a singleton list containing the subtree. To generate an and–or tree for a vector pattern p of length m that occurs in rule n, we produce an or–node containg the triple consisting of the vector length guard m, the singleton list n and a list containing, in order, the and–or trees generated from the subpatterns of p. To merge such a pattern with a tree t that is an or–node that includes a triple guarded by the vector length guard m, and m subtrees, add n to the triple's list of selected nodes, and merge the subpatterns of p with the corresponding subtrees of t. To merge such a pattern with a tree t that is an or–node that does not include a triple guarded by the vector length guard m, add a triple to t consisting of the vector length guard m, the singleton list containing n and a list of subtrees produced by generating and-or trees for each sub-pattern of p. The details of this design are probably a mistake. Instead, we might have introduced a new sort of node, the vector–or–node that would contain triples containing an integer length guard, a list of selected rule numbers, and a list of subtrees. This would have saved some space in or–nodes for ordinary constructor applications, (singleton lists have a three words of overhead, option cells only two words of overhead.) and would have avoided contaminating the lambda language with an extra sort of guard. The disadvantage of such a mechanism would have been extra cases in functions that manipulate and–or trees.

Abstract constructors are problematic because they can have arbitrary non-functional types, and so can appear anywhere in a pattern. For example, they can appear where an integer pattern is expected, or where a record pattern is expected. Moreover, since they are simply aliases for other values, they do not mesh well with the distinct cases paradigm provided by the lambda expression's case term.[1] Our solution is simple. We add a field for abstract constructor tests to every node. This field consists of a list of triples containing a constructor, a list of selected rules and an optional subtree. To generate an and–or tree for an abstract constructor pattern in rule n, produce a leaf–node with an abstract test field containing a triple that contains the pattern's constructor, the singleton list containing n, and no subtree. To merge an abstract constructor pattern k occurring in rule n with an existing tree t that already has a triple guarded by k in its list of abstract tests, simply add n to the list of selected patterns for this triple. To merge an abstract constructor pattern k occurring in rule n with an existing tree t that does not have a triple guarded by k in its list of abstract tests yet, simply add a triple containing k, the singleton list n and no subtree to the list of abstract tests. Abstract constructor applications are handled analogously.

Variables must be treated more carefully because they have an additional role in our setting. As well as marking the holes occupied by values to be bound in the right hand side, they also mark the holes occupied by pieces of values to be matched against by later constraints. These values may not always be available; for example, the cdr of a list, i.e., the second component of its deconsing, is not available for the empty list value. In Standard ML, this is not a problem, because the values are not required until a particular pattern has been completely matched. At this point, all values required by the bindings of the matched pattern are guaranteed to exist. In our setting, some of these values may be required during pattern matching (so that constraint match testing can be done). Our tree minimization heuristics work best when every safely executable test is considered a potential next test. Thus, it is important that constraint tests be turned into normal tests as quickly as possible. This in turn demands that we determine as quickly as possible the availability of the values against which constraints match. To do this, we record every node in the and–or tree at which a variable is bound. When the node is made available, we know that the value to which the variable is bound is available, and can notify the constraints of this fact. Thus we add to every field information about its binding of variables. To generate an and–or tree for a variable pattern, we produce a leaf–node that is marked as a binding node. To merge a variable pattern with

[1] Actually, the latter problem already exists in Standard ML because of exception aliases. This is a source of a bug in Standard ML of New Jersey. It also means that even for ordinary patterns the above and–or tree generation is slightly wrong.

an existing and–or tree, mark the root node of the tree as a binding node.

Layered patterns must be treated more carefully both because variables need to be, and also because abstract constructor patterns and abstract constructor application patterns may appear as their layering patterns. To generate an and–or tree for a layered pattern it suffices to generate the and–or tree for one of its sub-patterns and merge in the other pattern.

One and–or tree is produced for each path that appears as the tested value of a constraint. At least one is always produced, for the root path.

4 Pattern Flattening

Next we flatten the and–or tree into a list of available decisions. A decision may be a case decision, which includes a path to be tested, a list of possible guards (if this is finite), a list of case triples, and a list of don't care rule numbers. Or it may be an abstract constructor decision, which includes a path to be tested, the abstract constructor to be tested against, a list of the rule numbers of rules that require this decision to succeed, a list of the rules that do not require the decision to succeed, and a list of decisions that become available if the decision succeeds. Or it may be a binding decision, which simply gives a path to be bound. The case triples in a case decision consist of a guard constant, a list of selected rule numbers and a list of decisions that become active if the case matches.

This is done by recursive descent on the and–or tree. In addition to the current subtree the procedure that implements the conversion takes the current path. To convert an and–node at path p, this procedure calls itself recursively on each subtree of the and–node, using the ith projection of p as the path for the ith subtree. It flattens the resulting list of decision lists. To this resulting list, it adds a binding decision for p if the and–node binds. It also adds decisions for any abstract constructor tests associated with the and–node. To convert an or–node, a case decision is produced. Its path is the current path, it has one case triple for each element of the or–node. The list of possible guards is empty, unless the guards are datatype constructors, in which case the list contains all the constructors of the datatype. The don't care rules are those rule numbers that appear in no guard triple. To this are added any abstract constructor decisions associated with the node, and a binding decision if the node binds. leaf–nodes are translated as the the list of abstract constructor decisions associated with them plus a binding decision if they bind. The case triples consist of the guard constant k, the rules that this guard appears in, and the result of flattening its subtree with the path $\delta_k(p)$. The generation of abstract decisions also involves a flattening of any subtrees.

Two refinements are necessary. First, or–nodes that have vector length guards result in case statements that test the vector length of the current path, and have integer guards. Moreover, instead of flattening the subtrees of the cases with the path $\delta_k(p)$, they are flattened with the appropriate vector projection of p, and the result flattened. Second, if the only guard of the or–node is the only constructor k of a datatype, no case decision need be generated. Instead, we simply generate the decisions for the binding and abstract tests, and add them to the decisions produced by flattening the node's single subtree with the path $\delta_k(p)$.

The value to be tested by a given constraint pair is flattened to reveal the list of paths that must be available before the constraint's decisions can be made available. For the decisions arising from the and–or tree for the root path, this will yield simply the root path.

5 Decision Tree Generation

From the resulting list of patterns, it is an easy matter to generate a decision tree: pick the best available aplicable decision, generate the decision node for it. For each possible outcome of the decision, add any newly available decisions to the remaining decisions, and continue. When no applicable decisions remain, generate the appropriate dispatch for a right hand side. Binding decisions are always best, and never generate code. They may however make new decisions available by making constraint decisions available. For other decisions, the set of active rules is important. A decison is applicable only if at least one active rule does not appear in it's don't care set. Case decisions result in case nodes. There is one subtree for each case triple, plus one default tree if the case is inexhaustive. For a given case, the subtree is generated from the remaining decisions plus any decisions from the triple, using the active rules that appear either in the case triple or the don't care as the new active set. For the default case, the subtree is generated from the remaining tests, using the active rules that appear in the don't care list as the new active set. Abstract tests are treated analagous, except that they yield abstract test nodes instead, and always have two subtrees. The dispatch generated when there are no more applicable tests is a dispatch of the right hand side for the first rule still active.

When, because of binding decisions, all the paths required for a constraints tests to be made available have been bound, we add the tests to the set of available tests.

The heuristics used to select the best decision are very simple. First, binding decisions are selected. Second, if there are no binding decisions available, the decision with the fewest don't care rules is selected. Third, if the previous rule doesn't uniquely specify a decision, the decision with the fewest cases is selected from among those selected by the previous rule. If this rule still doesn't produce a unique selection, one of the selected decisions is chosen arbitrarily.

6 Binding paths to variables

Next, we annotate the decision tree with binding points. Correctness dictates that we must bind every destructuring path used, either by a case statement, or an abstract constructor test, or as a subpath of a destructuring path being bound. It also dictates that we may only bind a destructuring path at points in the tree where we know they refer to components of the value. For example, we cannot bind the destructuring path that de-conses the root destructuring path except in portions of the decision tree where we are guaranteed that the root destructuring path is a cons value. Runtime efficiency suggests that we should not bind a path at a node of the tree if the decision tree includes a branch from that node to a leaf that includes no use of the destructuring path. It also suggests that a given destructuring path be bound at most once on any branch from the root to a leaf of the tree. Code space efficiency suggests that a destructuring path be bound as few times as possible. Instruction scheduling considerations suggest that a destructuring path be bound as early as possible, and certainly well before it is used. All these suggestions suggest a heuristic for chosing binding positions. We bind a destructuring path just rootward of each node with these two properties. First, every branch from the node to a leaf of the tree includes a use of the destructuring path. Second, no node of the tree closer to the root binds the destructuring path. This heuristic optimizes runtime. It ensures binding before use because every branch from a node that uses a destructuring path to a leaf includes a use of that destructuring path. It ensures that paths are not bound too soon because uses of deconstructors in paths are always guarded by conditional tests (either case statements or abstract tests). Uses only occur in the subtree of these tests where the path is well defined, and the tests always have multiple subtrees. It somewhat satisfies the other considerations by placing bindings as close to the root of the tree as possible.

The annotation is performed by a recursive tree walk on the decision tree. The procedure that implements the steps of this walk takes as an argument the set of all destructuring paths that *must* have been bound higher up the tree (either because they are used, or because they were implicitly bound by an abstract constructor test). It returns the set of all destructuring paths used but not bound by the tree. At leaf nodes, this procedure simply returns the list of destructuring paths required by the right hand side dispatch. At a case node, this procedure adds the destructuring path of the value being tested by the case statement to the input set to form the set A, and calls itself recursively on each of the node's subtrees with the set A. This results in a collection of sets. The procedure takes their intersection, and forms the union of this set with A. Let B be the resulting set. For each subtree t, add, to the top of t, a binding for every destructuring path used but not bound by t that does not occur in B . Return B. The operation of this procedure is similar for abstract constructor tests. Note that for abstract constructor tests against unary constructors, we must account for the implicit binding performed by a successful test.

A slight complication arises because paths are structured objects, and the subpaths of a path must be bound before it is. We do two things to deal with this problem. First, we add every subpath of a destructuring path to a set when we add a path. Second, we calculate the length of each path (the root path has length zero, other paths have length one greater than the length of their longest subpath), and if multiple bindings are to be performed at a given point in the tree, shorter paths are bound closer to the root.

7 Generating code

It is now a simple matter to generate code from the annotated decision tree. A destructuring path used at a node is bound by an ancestor of the node. Binding nodes are translated to application terms that apply lambda terms; case nodes are translated to case terms; abstract constructor tests are translated to the appropriate function calls or equality tests. The translation is performed in a top down fashion by a recursive procedure that traverses the decision tree. This procedure also takes an incremantally constructed environment mapping destructuring paths to variables. Initially this environment contains only a variable assignment for the root destructuring path.

The result of this phase of match compilation is a lambda expression that performs case selection, value binding and right hand side dispatch. We wrap this lambda expression with local bindings for the right hand side functions, and produce a lambda term with the resulting term as its body and the variable used for the root destructuring path as its binding.