Patterns

We now enhance our sugared source λ-calculus to handle patterns. Formally, a pattern is one of:

- a variable `v`
- a constant `k` (such as a number, string, etc.)
- a tuple of patterns $(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ $(n \geq 0)$
- a constructor applied to a pattern `c(p)`
- a wildcard `_`

For simplicity, we require that no variable appears more than once in a given pattern. (This can affect the semantics in lazy languages.) Patterns must also be well-typed.

Examples

```
_
17
z
(2,w,TRUE)
FALSE
TREE(_,LEAF(x))
TREE(TREE(LEAF(x),LEAF(12)),TREE(z,_))
```

Some languages have additional features, such as **conditional** equations, which can be dealt with during pattern matching; we will ignore these.

Rules

A **rule** is a pattern and an associated expression. We write it like a λ-abstraction, with the pattern in place of the bound variable.

Examples

```
λTREE(LEAF(x),y).+(x, s y)
λLEAF(x).(x,x)
λBLUE.0
λ(x,y).y
λ_.false
λx.x+2
```

A rule acts roughly like a λ-abstraction. When it is applied, one of two things happens:

• The rule pattern matches the argument, in which case the application returns the rule body with its variables instantiated by the corresponding pieces of the argument.

• The rule pattern doesn't match, in which case the application returns the special "value" `FAIL`.

Patterns and Values

From now on, we'll assume an **eager** language. We can describe the values to which a rule will be applied as tree-shaped data structures (terms), namely one of:

- a constant (such as number, string, etc.)
- a tuple of terms `(`$\mathtt{t}_1$`,...,`$\mathtt{t}_n$`)` ($n \geq 0$)
- a constructor applied to a term `c(t)`

Note the similarity in structure between patterns and terms.

It is easy to give an inductive definition of what it means for a pattern to match a term:

- a wildcard pattern matches any term.
- a variable pattern matches any term and binds the variable to that term within the body of the rule.
- a constant pattern matches a constant term iff the constants are the same.
- a tuple pattern matches a tuple term iff the tuples are the same length (always true in a typed system) and corresponding members match.
- a construction pattern matches a construction term iff the constructors are the same and the arguments match.

Simple matching code

If we use the structured type representations previously presented, it is straightforward to generate core λ-code to match a value against a given pattern.

Example

```
datatype intlist = NIL | CONS of int * intlist
datatype bool = FALSE | TRUE
```

```
λ(CONS(x, NIL), FALSE).exp
```

becomes

```
λa.if BOXED(SELECT(0,a))
      andalso not BOXED(SELECT(1,SELECT(0,a)))
      andalso IEQL(SELECT(1,a),0)
   then let x = SELECT(0,SELECT(0,a))
        in exp
   else FAIL
```

(where `if`, `andalso`, etc. are just sugar for the equivalent core λ-calculus).

Matches

A **match (sequence)** is an ordered list of rules, written separated by vertical bars (`|`).

Examples:

```
( λLEAF(x).true
| λTREE(x,y).false )
```

```
( λ0.0
| λ1.1
| λx.+(x,1) )
```

A match also behaves much like an abstraction. When it is applied, a matching rule is selected from the list and the value of its body is returned; if no rule matches, `FAIL` is returned.

If more than one rule of a match is applicable (the match is **overlapping**), we must have some way to choose which rule to use. We will assume that the **first** matching rule in the sequence should be chosen (as in SML).

Some FL's use other criteria, such as choosing the **most specific** rule that matches ("best fit"). These are more complicated to implement, and we won't explore them.

Nonexhaustive and Redundant Matches

A match that contains rules to cover all possible arguments, and hence never returns `FAIL`, is said to be **exhaustive**. It is useful for the compiler to flag **nonexhaustive** matches, although they may be legitimate. If a match returns `FAIL` at runtime, an exception is raised or the program dies.

Nonexhaustive Example

```
( λ0.0
| λ1.1 )
```

An overlapping match may be **redundant**, i.e., there may be rules in the sequence that can never be matched because they are "masked" by rules earlier in the sequence. Redundant matches are always an error and should be flagged as such by the compiler (although SML gives only a warning).

Redundant Example

```
( λx.+(x,1)
| λ0.0
| λ1.1 )
```

Naive Pattern Matching

The simplest way to implement a match sequence is to generate matching code for each rule, and connect the resulting fragments into an if-then-else chain:

λarg.
if pattern$_1$ matches arg then
  rule-body$_1$
else if pattern$_2$ matches arg then
  rule-body$_2$
else ...
else if pattern$_n$ matches arg then
  rule-body$_n$
else `FAIL`

But this approach can be quite inefficient because information gained about the structure of the arg during a partially successful match is discarded before starting the next match.

Somewhat more trivially, it doesn't use information about the range of values allowed in a term position, and doesn't permit use of `SWITCH` expressions on constructor tags.

Example

```
datatype intlist = NIL | CONS of int * intlist
type boolean = FALSE | TRUE

( λ(NIL, FALSE).exp1
| λ(CONS(x, NIL), FALSE).exp2
| λ(CONS(y, z), FALSE).exp3
| λ(_, TRUE).exp4 )

λa.if not BOXED(SELECT(0,a))
      andalso IEQL(SELECT(1,a),0)
   then exp1
   else if BOXED(SELECT(0,a))
      andalso not BOXED(SELECT(1,SELECT(0,a)))
      andalso IEQL(SELECT(1,a),0)
   then let x = SELECT(0,SELECT(0,a))
        in exp2
   else if BOXED(SELECT(0,a))
      andalso IEQL(SELECT(1,a),0)
   then let y = SELECT(0,SELECT(0,a))
        and z = SELECT(1,SELECT(0,a))
        in exp3
   else if IEQL(SELECT(1,a),1)
   then exp4
   else FAIL
```

Given the argument `(CONS(1,NIL), TRUE)` it takes 7 tests and 8 selects, some repeated several times, to determine the correct match ($\texttt{exp}_4$).

Given `(CONS(1,CONS(2,NIL)), FALSE)` it takes 5 tests and 6 selects to determine the correct match ($\texttt{exp}_3$) and 4 additional selects to bind the variables `x` and `y`.

Smarter matching

We can do better by analyzing the **entire** match sequence at once, and generating run-time matching code that explores the argument term only once, and does so in an **efficient order.**

Of course, we must preserve the semantics of sequential match testing. Our goal is to solve the **dispatching problem**, which may be formulated as follows:

Given a sequence of patterns $\texttt{p}_1,\ldots,\texttt{p}_n$, find out in which order the subterms of any possible argument term `t` must be examined to determine, with the minimum number of tests on the subterms of `t`, the smallest i for which $\texttt{p}_i$ matches `t`.

Example Revisited

```
( λ(NIL, FALSE).exp1
| λ(CONS(x, NIL), FALSE).exp2
| λ(CONS(y, z), FALSE).exp3
| λ(_, TRUE).exp4 )

λa.switch (SELECT(1,a))
   case 0:
      let a0 = SELECT(0,a)
      in if BOXED(a0)
         then if BOXED(SELECT(1,a0) then
                let y = SELECT(0,a0)
                and z = SELECT(1,a0)
                in exp3
              else
                let x = SELECT(1,a0)
                in exp2
         else exp1
   case 1: exp4
```

(where `switch` and `case` are sugar for the core `SWITCH` expression)

Now `(CONS(1,NIL), TRUE)` can be matched in just one select and test!

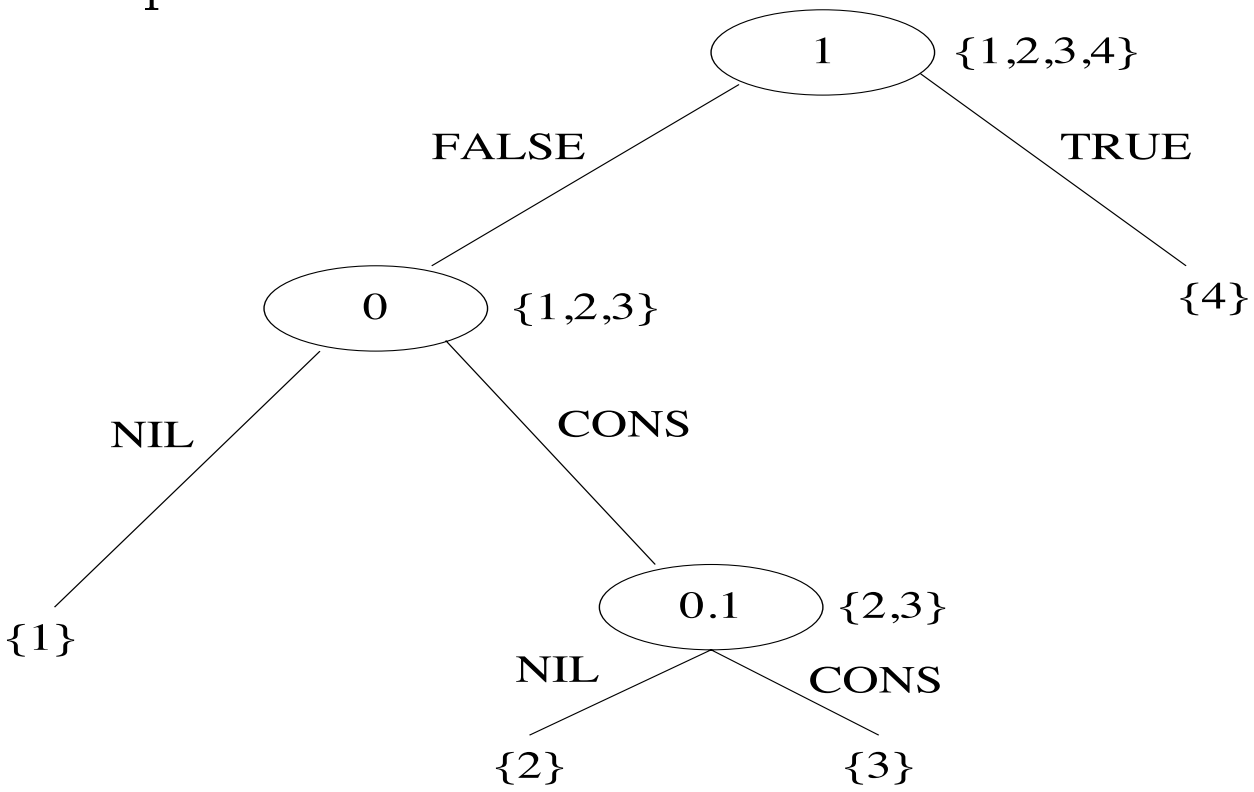
`(CONS(1,CONS(2,NIL)), FALSE)` can be matched in 3 tests and 3 selects, and the variables can be bound in just 2 further selects.

Formalizing the Dispatching Problem

The code we generate for a match is essentially a **decision tree.** The tree's nodes are labeled with the subterms of the argument term that are to be examined to find the correct match. The branches coming out of a node are labeled with the possible results of testing that subterm, i.e., the constructors or constants for that term.

Executing the match code corresponds to traversing the tree from root to a leaf, executing the tests corresponding to the nodes encountered along the path. We annotate each node with the set of patterns that are still **live** candidates after traversal has reached that node.

Example



Building Trees

We can build a decision tree for a given match in top-down fashion. At each stage in the construction we choose a **live subterm** to test. Intuitively, a live subterm is one that is guaranteed to exist if we have traversed this far in the tree. In our example, the initially live subterms are fields 0 and 1 of the pair. After we have determined that field 0 contains a `CONS` pair, both fields of that pair become live subterms.

For each subterm, there is a corresponding set of subpatterns drawn from the original pattern list. We exclude a subterm from the set of live subterms if all the corresponding subpatterns are variables or wildcards, since there can be no point in testing the subterm.

Subterms representing constructions are tested with a `SWITCH`; the number of branches is the number of constructors for the type. Numeric subterms are tested against specific constants mentioned in the pattern sequence, and have two outcome branches.

A pattern remains live only along the branch matching the corresponding subpattern constant. Any patterns whose corresponding subpattern is a variable or wildcard remain live along every outcome branch.

More on Building

To continue the construction process we follow each branch out of the test, recording the live patterns and live subterms along that branch.

• If there is only one live pattern, and no live subterms to test, we've reached a leaf: the matching process should terminate by choosing the one pattern.

• If there are multiple live patterns, but no live subterms to test, the match is overlapping. We've reached a leaf, and the matching process should terminate by choosing the **lowest-numbered** live pattern; this guarantees that we obey the original match semantics.

• If there are no live patterns, the match is nonexhaustive; We've reached a leaf; the matching process should terminate with `FAIL`.

• Otherwise, we choose a live subterm, create a new test node for it as described above, and repeat for each of *its* result branches.

Once the entire tree is constructed we can check whether every pattern is selected at some leaf; if not, the pattern is redundant.

Optimal Trees (Baudinet and MacQueen, 1985)

In general, we can build many trees for a given match, depending on our choice of live subterm at each construction step. Our goal is to build one such that any argument is matched with as few tests as possible. This corresponds to minimizing the length of all paths from the root to a leaf, which we could attempt to do in a worst-case or average sense.

Instead, we use a slightly different definition of optimality: an **optimal tree** for a match is one with as few total nodes as possible. This corresponds to generating the most compact possible match code, and usually also minimizes path length.

The example tree is optimal in this sense (and also minimizes worst-case path length).

(We could also attempt to minimize selects, but number of selections is usually also well-correlated with number of test nodes.)

Thus, we can reformulate the dispatching problem as: build an optimal decision tree for a given sequence of patterns.

Unfortunately, this problem is NP-complete, so it is probably impractical to solve it exactly. Instead, we look for practical **heuristics** that can be used to select from the set of live subterms as we construct a decision tree, and that lead to an optimal tree in most cases.

Relevance

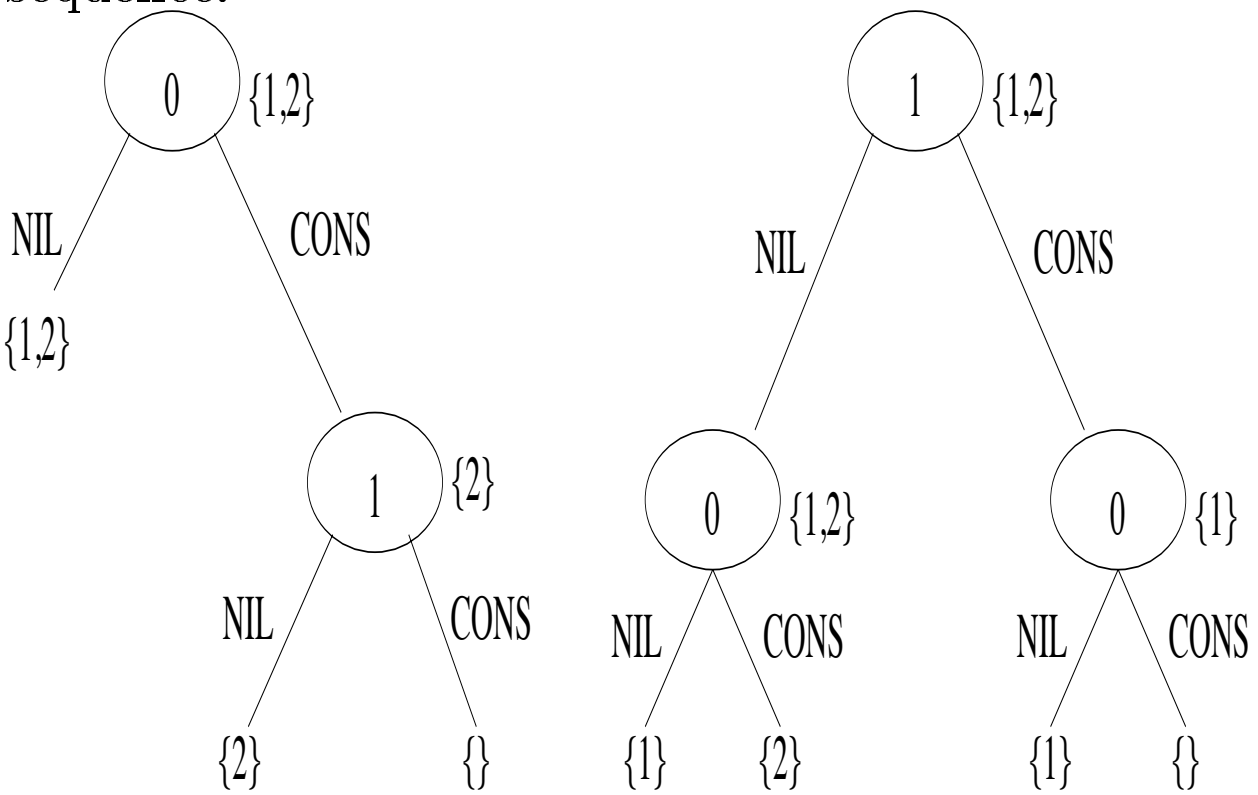
At each stage in the construction of the tree, we have to choose which of one or more subterms to test next. Each heuristic prunes the set of subterms to choose from. Heuristics are applied in (some) order until only a single choice is left, or we run out of heuristics and have to make an arbitrary choice.

The **relevance heuristic** tries to select subterms that can pinpoint the **first** matching pattern in the match sequence as early as possible.

Example

```
( λ(NIL,x,z).e1
| λ(y,NIL,z).e2 )
```

There are only two plausible decision trees for this sequence.



More relevance

The first tree is better because, by testing field 0 first, it isolates pattern 1 more quickly as a leaf of the tree, and pattern 1 must win over pattern 2 in case of ambiguity. Testing field 1 doesn't help isolate pattern 1, because pattern 1 is still a possible solution regardless of the outcome of the test. We say testing field 0 is **relevant** to pattern 1, but testing field 1 is **not relevant** to pattern 1. Conversely, testing field 1 is relevant to pattern 2 but testing field 0 is not. Testing field 2 is not relevant to either pattern.

In general, a test on a subterm is **relevant** to a pattern $\mathtt{p}_i$ iff i does not appear in the set of live patterns for at least one out-branch from the test node.

Given a set of live subterms to test and a set of live patterns, the relevance heuristic returns the subset of tests which are relevant to the **lowest-numbered** pattern for which any test is relevant. If this subset is a singleton, its element is used as the next test.

Otherwise, we try another heuristic, such as...

Branching Factors

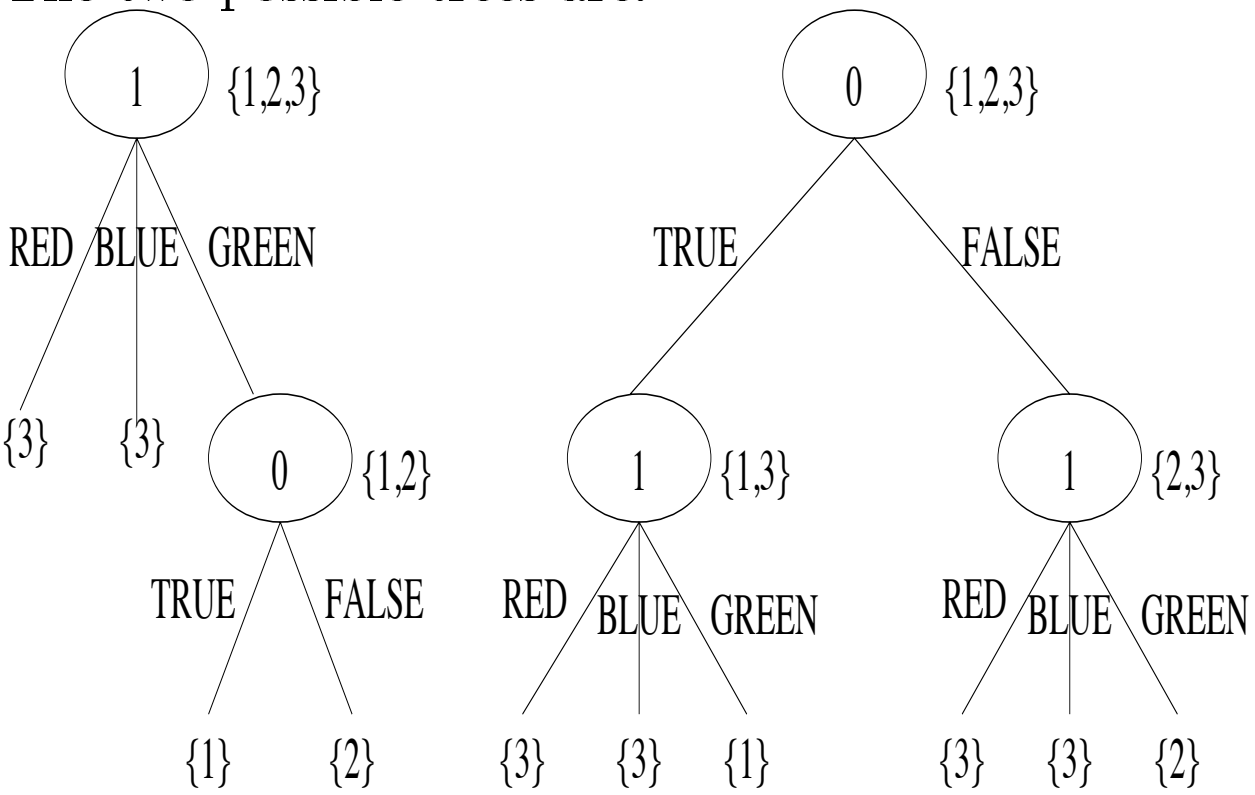
Idea: minimize size of tree by choosing nodes with fewest non-leaf successors.

Example

```
datatype color = RED | GREEN | BLUE
```

```
( λ(TRUE, GREEN).e1
| λ(FALSE, GREEN).e2
| λ_.e3 )
```

The two possible trees are:



Other things being equal, it is better to test field 1 (with a branching factor of 1) before testing field 0 (with a branching factor of 2).

There are lots of other possible heuristics...