

Pattern Matching

CS 502

Lecture 3

8/28/08

Tree Pattern Matching

- ML functions can be defined by a sequence of pattern expression pairs called rules that are matched to a subject pattern.
 - The first rule that matches the subject has its expression evaluated and returned as the result of the pattern match.
-

Patterns

- Patterns can be either
 - Simple (variables, constants, wildcards)
 - Tuples of patterns
 - A constructor operation applied to a tuple of patterns.
 - This definition implies that patterns are tree-shaped built using constructors, tuples and simple values.
 - Goal: minimize the number of tests that have to be applied to determine the first rule that matches a subject.
 - For ML, this is an NP-complete problem.
 - Use heuristics that work well in practice.
-

Classical Approach

- Build a decision tree:
 - The tree encodes the pattern and defines the order in which subterms of a term have to be evaluated.
 - It also injects runtime tests to determine if a pattern matches a value.
 - Two classifications:
 - Exhaustive: there exists at least one pattern that matches the subject.
 - Redundant: there is more than one pattern that matches the subject.
 - Desire exhaustive non-redundant pattern expressions.
-

Example

fun diff(nil,y) = y (1)

| diff(x,nil) = x (2)

| diff(hdx::tlx,hdy::tly) = diff(tlx,tly)

- ❑ The set of patterns is exhaustive
 - ❑ Patterns (1) and (2) are non-disjoint or overlap because there is at least one value for which both patterns match (i.e., (nil,nil))
 - ❑ Ordering of rules impose a priority ordering among patterns.
 - ❑ Patterns must be linear: no variable in a pattern can occur more than once:
 - fun equal(x,x) = true (* this is illegal *)
 - ❑ Patterns should not be redundant: they should match some value, not matched by earlier patterns.
-

Syntax of patterns and terms

$p ::= _$ (wildcard)
| v (variable)
| c (constant)
| (p_1, \dots, p_n) (tuples)
| $C p'$ (constructor C applied to pattern p')

□ Patterns must be well-typed

- $\text{Cons}(3)$ not well-typed
 - What about $\text{Cons}(x)$?
-

Example

Value term $t = (\text{true}, \text{Cons}(1, \text{Nil}))$

Following sequence of patterns of type $(\text{bool} * \text{intlist})$:

$p1 = (\text{true}, _)$

$p2 = (\text{false}, \text{nil})$

$p3 = (\text{false}, \text{Cons}(x, \text{Nil}))$

$p4 = (\text{false}, \text{Cons}(y, z))$

$p1$ agrees with t on its first component.

$p2, p3, p4$ do not.

$p1$ also agrees on its second term.

$p3$ and $p4$ have a pattern of the form $\text{Cons}(p')$ and agree with t on this component.

$p3$ and $p4$ also agree with t on both arguments to Cons .

Formal Definition

Given pattern p and value term t , p agrees with t on a subterm t^* of t iff:

$p = _$ (wildcard)

p is a variable

$p = c$ where c is a constant and $t = c$

$p = (p_1, \dots, p_n)$ $t = (t_1, \dots, t_n)$ and $t^* = t$

$p = (p_1, \dots, p_n)$, $t = (t_1, \dots, t_n)$, t^* is a subterm of t_j for $1 \leq j \leq n$ and p_j agrees with t_j on subterm t^*

$p = C p'$ and $t^* = t = C t'$

$p = C p'$, $t = C t'$, t^* is a subterm of t' , and p' agrees with t' on t^* .

Matching

A pattern p matches a term t iff p agrees with t on all subterms of t .

The matching problem:

Given a finite sequence of patterns p_1, \dots, p_n and value term t , find the first pattern in the sequence which matches t , if one exists. (We elide issue of typing here.)

Efficiency

- Naive approach:

- Match each argument with each pattern in turn (top-down), starting over after every failure.
- Inefficient because information gleaned about the argument in each partially successful match is ignored on subsequent matches.

- Better strategy: analyze pattern sequence at compile-time, and try to minimize number of case discrimination tests.

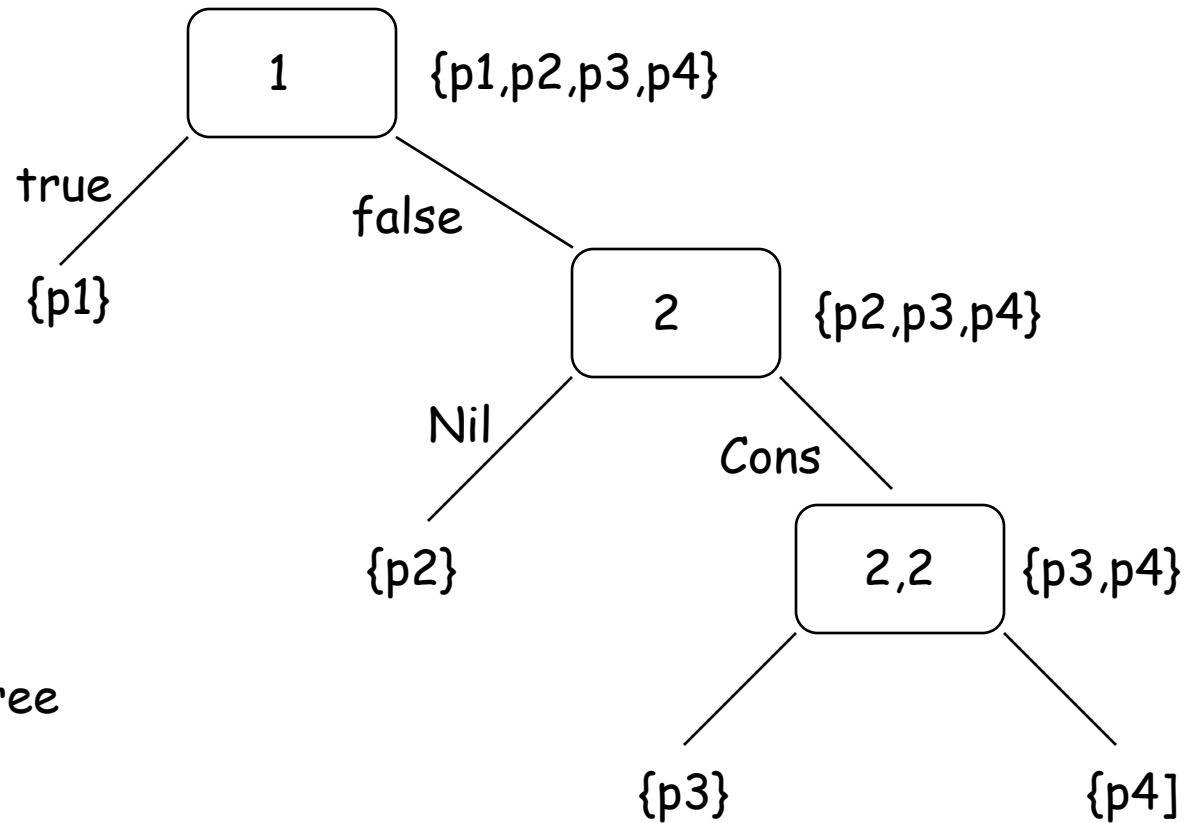
Example (revisited)

- Term $t = (\text{true}, \text{Cons}(1, \text{Nil}))$
 - Patterns:
 - $p1 = (\text{true}, _)$
 - $p2 = (\text{false}, \text{Nil})$
 - $p3 = (\text{false}, \text{Cons}(x, \text{Nil}))$
 - $p4 = (\text{false}, \text{Cons}(y, z))$
 - Want to find the pattern that matches t . Must compare subterms of t with subpatterns.
-

Example (revisited)

- Two choices: test either on first component or second component.
 - First component: discover that p1 is the only pattern that matches. Wildcard in second component establishes p1 as a matching pattern.
 - Second component: can discard p2. Must compare against p1, p3, and p4:
 - Three additional tests required:
 - Test on first component of t. (leads to desired result)
 - Test on first component of Cons.
 - Test on second component of Cons.
-

Decision Tree



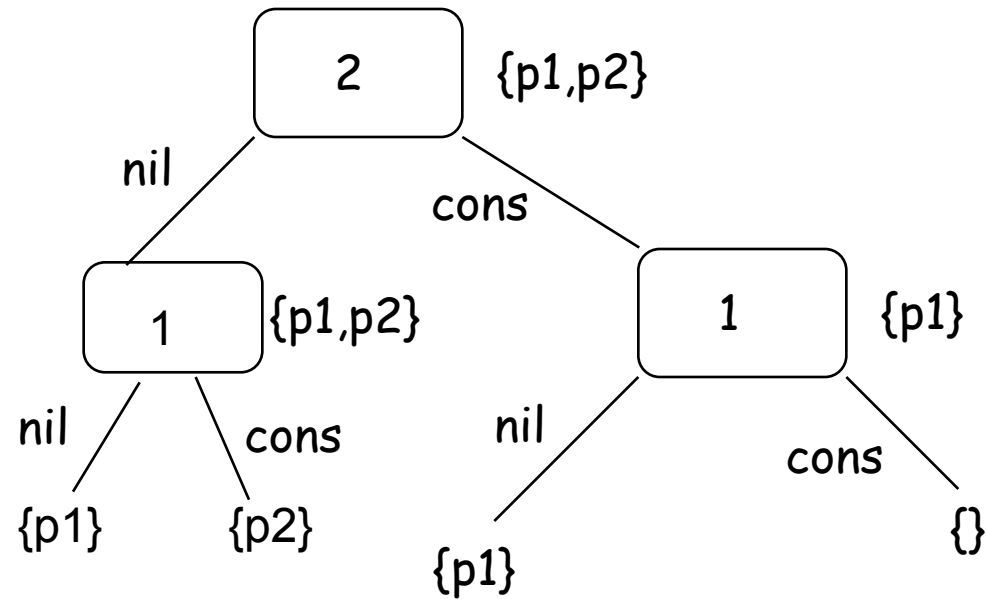
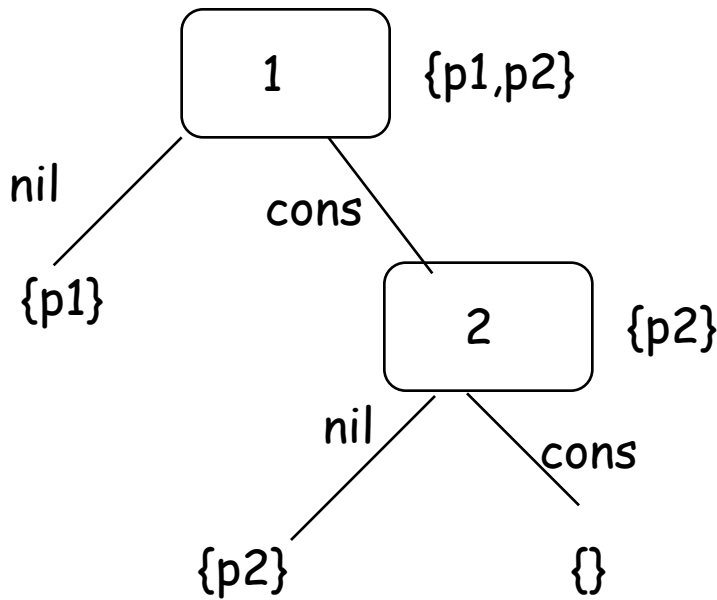
Minimal decision tree
for example

Finding Optimal Decision Trees

- Finding the optimal (minimal) decision tree is NP-complete.
 - In practice, employ a variety of heuristics to achieve this goal.
 - Different compilers employ different schemes to reduce the size of the decision tree constructed under the naive interpretation.
-

Relevance Heuristic

- If two patterns overlap, the one with the lowest index is preferred.
- Consider (p1) (nil, x) and (p2) (y, nil)



Branching Factor Heuristic

- Favoring choice of tests with low branching factor first.
- `type color = red | blue | green`
- `(p1) (true, green), (p2) (false, green)`

