Robust Scripting via Patterns

Bard Bloom and Martin Hirzel
IBM T. J. Watson Research Center
October 2012
Setting

• Thorn language
  • IBM and Purdue project, now in stasis

• Dynamic Languages
  – No static types

• Concrete Data Structures
  – Lists, records, objects / datatypes

• Imperative languages
  – But emphasis on declarative/functional
Related Work

• SNOBOL4 (1966)
• ML, ISWIM, Hope, Haskell, F#, Scala, Kotlin
• Scheme, Newspeak, Python, Converge, OMeta
• OCaml, JMatch
• Views, Tom, Matchete
Plan

• Pattern Language
  – Some fancy patterns
  – First-class Patterns

• Integration with Thorn
  – Patterns used everywhere
  – Some interactions with standard control flow

• Usage
  – Do Thorn programmers do what they can do?
Patterns (in the ML Sense)

- Match a **subject** value against a **pattern**
  - Can **FAIL**
  - Can **SUCCEED** and bind some **variables**

<table>
<thead>
<tr>
<th>Name</th>
<th>Subject</th>
<th>Pattern</th>
<th>Result</th>
<th>Bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>[1,2,3]</td>
<td>x</td>
<td>succeed</td>
<td>x=[1,2,3]</td>
</tr>
<tr>
<td>List</td>
<td>[1,2,3]</td>
<td>[x]</td>
<td>fail</td>
<td></td>
</tr>
<tr>
<td>Wildcard</td>
<td>[1,2,3]</td>
<td>[x,<em>,</em>]</td>
<td>succeed</td>
<td>x=1</td>
</tr>
<tr>
<td>Head/Tail</td>
<td>[1,2,3]</td>
<td>[x, y...]</td>
<td>succeed</td>
<td>x=1, y=[2,3]</td>
</tr>
<tr>
<td>Literal</td>
<td>[1,1]</td>
<td>[x, 1]</td>
<td>succeed</td>
<td>x=1</td>
</tr>
<tr>
<td>Value</td>
<td>[1,1]</td>
<td>[x, $x]</td>
<td>succeed</td>
<td>x=1</td>
</tr>
<tr>
<td>Record</td>
<td>&lt;a=1,b=2,c=3&gt;</td>
<td>&lt;a=x, b&gt;</td>
<td>succeed</td>
<td>x=1, b=2</td>
</tr>
</tbody>
</table>
How Much Are They Used?

• Corpus:
  – 24K lines of code
  – Most of the Thorn code in existence

• Coders
  – Bard (60%), skilled (30%), novices (10%)

• Purposes
  – Some examples of Good Thorn Style
  – Some one-shot programs to throw away

• **This Is Not Science**
  – Literary Analysis, maybe

• Negative results may be interesting too
Part I: Control and Patterns
Control Structures and Patterns

• **Design Principle:** Put patterns wherever they might make sense

• **Design Principle:** Patterns should be allowed wherever variables are bound to arbitrary values
  – If it makes sense
  – Deal with failure somehow
  – *E.g.* Formal parameters can be patterns
Binding Statement

• Binding statement (LISP/ML let):
  – \texttt{x}=[1,2,3]

• With pattern, it's destructuring
  – \texttt{[a,b,c] = [1,2,3]}
  – Exception if fails

• Usage: 3\% of bindings have interesting pattern
  – Bard prefers defensive programming
Scopes

• **Design principle:** pattern matches introduce variables into the scope that will be executed iff the match succeeds.

• **Match Operation:** \( E \sim P \)
  - returns true on success, false on failure
  - Produces bindings in right scope

• But what's the right scope?
  - Depends on context...
if statement

• if(L ~ [x])
  use(x);
else
  xUndefined();
• We support
  if (A ~ P && B ~ Q && C ~ R)
  – (But not general propositional logic)
• 37% of if's have matches
• (There's a match statement too, but much less used than 'if')
Patterns and while

- while: bindings in test can be used in body

```plaintext
while(R ~ <x>)
   R := munge(R,x);
   xUndefined();
```
Patterns and until

• Until: bindings in test can be used after body
  – until(x.spouse ~ (!null && y))
    x.date();
    fileJointly(x,y)
  – Precisely expresses "look for something"

• Rarely used (<1%)
  – Searching comprehensions and recursion are favored.
  – Thorn bias: Most whiles were while(true) in actor bodies
Patterns and Control, reprise

• There's value to making patterns aware of control:
  – if, for: 40%
  – fun, lambda: 20%
  – let, while, until: 1-3%
Part II: Fancy Patterns
Kinds of Patterns

• Common Patterns
  – Most patternly languages have these
  – wildcard, variable, literal, list, ...
  – 82% of Thorn patterns are common
    • Count of syntax tree nodes
    • Not counting variables

• Fancy Patterns
  – Few languages have any of these
  – Fewer have all of them.
  – 18% of Thorn patterns are fancy
  – Let's see a couple...
Fancy Pattern: Type Test

• **General form:** $P : T$
  – matches a value of type $T$
  – which must also match pattern $P$
  – And binds what $P$ does

• **Idiom:**
  – `fun f(x:int) = x+3;`

• **Usage:** 3.5% of all patterns
## Fancy Pattern: Boolean Combinations

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
<th>Binds</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>P &amp;&amp; Q</td>
<td>if both P and Q match</td>
<td>Everything bound by P or Q (disjoint)</td>
<td>3%</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td>Q</td>
<td>if either P or Q matches</td>
</tr>
<tr>
<td>!P</td>
<td>if P fails</td>
<td>nothing</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
&& is useful

- **Pattern:** \( x && [y, z...] \)
  - Matches a nonempty list
  - Binds the whole list \( (x) \), the head \( (y) \) and tail \( (z) \)

- **as** construct in pattern-bearing languages
  - "Get a whole value and its parts"

- **Trans-as usage:**
  - \([..., 1, ...] && [..., 2, ...]\)
  - Matches a list containing 1 and 2 in either order

- **About 3% of patterns involve &&**
  - Mostly for the as idiom.
  - No popular idioms for || and !
  - A good idiom makes a pattern operator popular.
Internal Matches

• General Form: E~P
  – Succeeds if value of E matches P
  – Binds what P does
  – Can appear inside of patterns
  – Usage: 3.5%

• Example: \([x] \&\& f(x) \sim [y,z]\)

• Swiss Army Construct
  – E.g. optional field foo, defaulting to 22:
    \(<\text{foo}=x> \mid | \mid 22\sim x\)
Part III: First-Class Patterns

• Fanciest of all the fancy patterns.
First-Class Patterns

• First-class functions are amazingly useful
  – One of the top N ideas in programming languages
First-Class Patterns

• First-class **functions** are amazingly useful
  – One of the top N ideas in programming languages

• First-class **patterns** are a bit cool
  – One of the top $N^3$ ideas in programming languages
Why abstract patterns?

• Summing binary trees

• Object/datatype representation:
  
  \[
  \text{fun sum(} \text{Fork(l,x,r)) = sum(l) + x + sum(r);} \\
  \mid \text{sum(Leaf(x))} \quad \text{= x;}
  \]

  (This is the nicest code in the universe)
Why abstract patterns?

• Summing binary trees

• Object representation:
  \[
  \text{fun sum(Fork(l,x,r))} = \text{sum(l)} + x + \text{sum(r)}; \\
  \text{|sum(Leaf(x))} = x;
  \]

• List representation:
  \[
  \text{fun sum([l,x,r])} = \text{sum(l)} + x + \text{sum(r)}; \\
  \text{|sum([x])} = x;
  \]

(this is also the nicest code in the universe)
Why abstract patterns?

• Summing binary trees

• Object representation:
  \[
  \text{fun } \text{sum}(\text{Fork}(l,x,r)) = \text{sum}(l) + x + \text{sum}(r);
  \]
  \[
  \text{    |sum(x)} = x;
  \]

• List representation:
  \[
  \text{fun } \text{sum}([l,x,r]) = \text{sum}(l) + x + \text{sum}(r);
  \]
  \[
  \text{    |sum(x)} = x;
  \]

• Are we not computer scientists?
  – And do we not abstract reflexively?
Pattern Expression, part 1

• Pattern Abstraction:
  - A value (not a pattern).
  - \text{pat} \ [x, y] = [x, $x$, y]
  - \(x,y\) are outputs not inputs.
  - \(x,y\) are scoped inside the expression

• Pattern Application
  - \(E[r,s]\) is a pattern
  - \(r,s\) are subpatterns
  - Appears in pattern context: \(\text{somelist} \sim E[r,s]\)

\[
E = \text{pat}[x,y] = [x, $x$, y] \\
L = [3, 3, 4] \\
\text{if (L } \sim E[a,b]) \text{ assert(a}=3, \text{ b}=4) \\
\text{if (L } \sim E[a,9]) \text{ fails()}
\]
Sum with Representation Parameter

- Representation pattern
  \[ rp = \langle \text{fork=fpat}, \text{leaf=lpat} \rangle \]
  - \[ rp\.fork[l,x,r] \] matches a fork node
  - \[ rp\.leaf[x] \] matches a leaf node

- Sum with explicit \( rp \):

  \[
  \text{fun sum2}(rp, rp\.fork[l,x,r]) = \begin{cases} 
  \text{sum2}(rp, l) + x + \text{sum2}(rp, r) & \text{if } x \neq 0 \\
  \text{sum2}(rp, \text{leaf}[x]) & \text{if } x = 0
  \end{cases}
  \]

No longer the most beautiful code in the universe
Computing the Representation

// Guess representation of a tree...
fun rep([_,_,_] || [])          = repList;
| rep(["Fork" || "Leaf", _...]) = repTaggedList;
| rep(x:Tree)                   = repTree;
| rep(<left,item,right> || <leaf>) = repRecord;

// Use it!
fun sum(rep(it).fork[l,x,r]) = sum(l) + x + sum(r);
| sum(rep(it).leaf[x])         = x;
Pattern Abstractions, parts 2-N

• More variations
  – pattern/constructor duality
  – inputs and outputs

• Late addition to language
  – We didn’t get to use them much

• Nice new toy!
Conclusion

• There's a lot more to patterns than ML-style
  \(-P\&\&Q, E\sim P, \text{pat}[x]=P\)
• Patterns can be meshed with statements
  \(-\text{if}(L\sim[x,y]) \text{ use}(x,y)\);
• If you have them, they will be used
  \(-\text{happily!}\)