Guarded Patterns

Mark P Jones
OGI School of Science & Engineering
Oregon Health & Science University

Evaluation in λ-calculus:

- To evaluate a λ-calculus term M:
  - Pick a redex: write M = C[R]
  - Rewrite it: C[R] B C[R'], if R B R'
  - Repeat until done ...

- A simple, two-stroke process.

Evaluation in Haskell:

- Complex language ⇒ complex process:

multiple equations

pattern matching

guards

```
filter p []                           = []
filter p (x:xs) | p x                 = x : rest
                | otherwise = rest
                  where rest = filter p xs
```

more complex patterns:
v@pat, ~v, literals,
(n+k), $C\{l_2=p_2,l_1=p_1\}$,
pat!f, views, ...

local definitions

This Talk:

An intermediate language that is:

- Flexible enough to allow a simple encoding of high-level constructs (and perhaps some future extensions)
- Simple so that it is easy to understand and use
- Easy to translate into lower-level constructs

Pattern Guards:

- A proposal by Simon Peyton Jones, dating back to at least 1997, but presented formally in a paper at the Haskell workshop 2000

- Recognizes a common case in which standard Haskell pattern-matching is quite “clunky”

- ... and suggests a generalization of guards that can help.

- Implemented in GHC, but not in any other Haskell implementations (as yet)

A clunky example:

Suppose that we have a function:

lookup :: FiniteMap $\rightarrow$ Int $\rightarrow$ Maybe Int

and consider the following definition:

```
clunky env var₁ var₂
      | ok₁ && ok₂  = val₁ + val₂
      | otherwise   = var₁ + var₂
       where m₁      = lookup env var₁
             m₂      = lookup env var₂
             ok₁     = isJust m₁
             ok₂     = isJust m₂
             Just val₁ = m₁
             Just val₂ = m₂
```

Surely there's a neater way to write this?

Cases don't help much:

We could rewrite it as:

$$
\begin{array}{l}
\text{clunky env var}_1\ \text{var}_2 \\
\quad = \textbf{case}\ \text{lookup env var}_1\ \textbf{of} \\
\qquad \text{Nothing} \ \rightarrow \text{var}_1 + \text{var}_2 \\
\qquad \text{Just val}_1 \rightarrow \textbf{case}\ \text{lookup env var}_2\ \text{of} \\
\qquad\qquad\qquad \text{Nothing} \ \rightarrow \text{var}_1 + \text{var}_2 \\
\qquad\qquad\qquad \text{Just val}_2 \rightarrow \text{val}_1 + \text{val}_2
\end{array}
$$

This is a little shorter, but it's not much clearer because it forces us to use lower level tools ...

Simon's Proposal:

Allow pattern matching within guards:

```
clunky env var₁ var₂
  | Just val₁ ← lookup env var₁,
    Just val₂ ← lookup env var₂ = val₁ + val₂
  | otherwise                   = var₁ + var₂
```

Guards are no longer just simple boolean expressions ... they can contain multiple entries, including generators of the form pat $\leftarrow$ exp, and local definitions of the form **let** decls.

Blending Matching & Evaluation:

	Left of "|"	Right of "|"
Haskell	match	test
Pattern Guards	match	match, test
Guarded Patterns	match, test	match, test

Guarded Patterns:

- A <u>guarded pattern</u> is a pattern of the form: (pat | quals) where quals is a list of qualifiers:
  - pat ← exp
  - **let** decls
- Boolean guards are just a special case:

```
filter p (x:xs)
        | True ← p x         = x : filter p xs
        | True ← otherwise = filter p xs
```

Syntax:

e ::= v
| $e\ e'$
| λm
| ...

p ::= v
| $C\ p_1 \ldots p_n$
| $(p \mid q)$

q ::= **let** decls
| $p \leftarrow e$

m ::= Commit e
| m Y m'
| FAIL
| $p \rightarrow m$
| $q \Rightarrow m$

Other Patterns as Sugar:

$\sim p$	$\equiv$ (u | **let** p = u) (u new)
v@p	$\equiv$ (v | p $\leftarrow$ v)
(n+k)	$\equiv$ (u | u$\geq$ k, n $\leftarrow$ (u-k))
lit	$\equiv$ (u | u==lit)
$C\{l_2=p_2,l_1=p_1\}$	$\equiv$ $(C\ u_1\ u_2 \leftarrow u \mid p_2 \leftarrow u_2,\ p_1 \leftarrow u_1)$
[(x,x)]	$\equiv$ ((x,x') | x==x') : []
p!f	$\equiv$ (u | p $\leftarrow$ f u)
(Im y)@(Ph th)	$\equiv$ (u | y$\leftarrow$getIm u, th$\leftarrow$getPh u)

N.B. For an IL, the (u | ...) form alone suffices

Example:

- Our old friend, filter:

```
filter p []                    = []
filter p (x:xs) | p x          = x : rest
                | otherwise = rest
                  where rest = filter p xs
```

- Representation:

```
filter = λ(p → [] → Commit [])
        Y(p → (x:xs) →
           let rest = filter p xs ⇒
            ((p x ⇒ Commit (x:rest))
           Y(otherwise ⇒ Commit rest)))
```

Semantics:

Pattern matches:

- $(p|q) \to m \qquad = p \to (q \Rightarrow m)$
- $(p \to m)\ e \qquad = (p \leftarrow e) \Rightarrow m$
- $(m_1\ Y\ m_2)\ e \qquad = (m_1\ e)\ Y\ (m_2\ e)$
- $(q \Rightarrow m)\ e \qquad = q \Rightarrow (m\ e)$

Fatbar, Commit, and Fail:

- $(m_1\ Y\ m_2)\ Y\ m_3 = m_1\ Y\ (m_2\ Y\ m_3)$
- Commit e Y m = Commit e
- FAIL Y m = m

Semantics (continued):

Guarded matches:

- $(C\ p_1\ \ldots\ p_n \leftarrow C\ e_1\ \ldots\ e_n) \Rightarrow m$
  $= (p_1{\leftarrow}e_1) \Rightarrow \ldots \Rightarrow (p_n{\leftarrow}e_n) \Rightarrow m$
- $(C\ p_1\ \ldots\ p_n \leftarrow D\ e_1\ \ldots\ e_m) \Rightarrow m$
  $=$ FAIL
- $(v \leftarrow e) \Rightarrow m \quad = (\textbf{let}\ v = e) \Rightarrow m$
- $((p \mid q) \leftarrow e) \Rightarrow m \quad = (p \leftarrow e) \Rightarrow (q \Rightarrow m)$

Distributivity:

- $(p \rightarrow m_1)\ \Upsilon\ (p \rightarrow m_2) \ = p \rightarrow (m_1\ \Upsilon\ m_2)$
- $(q \Rightarrow m_1)\ \Upsilon\ (q \Rightarrow m_2) \ = q \Rightarrow (m_1\ \Upsilon\ m_2)$

Highlights:

- Simple, expressive intermediate language;
- Haskell definitions are easy to rewrite in this notation, guarded patterns might be useful as a source-level construct;
- Pattern matching algorithm correctness follows from equations;
- Monadic type system + semantics;
- Algebraic approach facilitates reasoning.