Desugaring List Comprehensions and Pattern Matching

Arvind
Laboratory for Computer Science
M.I.T.

October 7, 2002

http://www.csg.lcs.mit.edu/6.827

Infinite Data Structures

1.
```
ints_from i  = i:(ints_from (i+1))

nth n (x:xs) = if n == 1 then x
               else nth (n - 1) xs
   nth 50 (ints_from 1) -->            ?
```

2.
```
ones = 1:ones
   nth 50 ones  -->                    ?
```

3.
```
xs = [ f x | x <- a:xs]
   nth 10 xs    -->                    ?
```

Primes: *The Sieve of Eratosthenes*

```
primes = sieve [2..]

sieve (x:xs) = x:(sieve (filter (p x) xs))

p x y = (y mod x) ≠ 0

      nth 100 primes
```


Desugaring!

- Most high-level languages have constructs whose meaning is difficult to express precisely in a direct way
- Compilers often translate ("desugar") high-level constructs into a simpler language
- *Two examples:*
  - *List comprehensions:* eliminate List compressions usings maps etc.
  - *Pattern Matching:* eliminate complex pattern matching using simple case-expressions

List Comprehensions

List Comprehensions: Syntax

`[ e | Q ]` where `e` is an expression and `Q` is a list of generators and predicates

There are three cases on `Q`

1. First element of `Q` is a generator

   `[ e | x <- L, Q’ ]`

2. First element of `Q` is a predicate

   `[ e | B, Q’ ]`

3. `Q` is empty

   `[ e | ]`

List Comprehensions Semantics

Rule 1.1 `[ e | x <- [], Q ]` ⇒

Rule 1.2 `[e | x <- (`e_x `:` e_{xs} `), Q ]` ⇒

Rule 2.1 `[ e | False, Q ]` ⇒

Rule 2.2 `[ e | True , Q ]` ⇒

Rule 3 `[ e | ]` ⇒

Desugering: *First Attempt*

TE[[`[ e | ]`**]]** = `e :[]`

TE[[`[ e | B, Q]`**]]** =
`if` `B` *`then`* **TE[[**`[e | Q]`**]]** *`else`* `[]`

TE[[`[ e | x <- L, Q]` **]]** =

L9-9
Arvind

Eliminating Generators

```
[ e | x <- xs] ⇒ map (\x-> e) xs

[ e | x <- xs, y <- ys] ⇒
```

where **concat** flattens a list:

```
concat[]       = []
concat (xs:xss) = xs ++ (concat xss)
```

```
[ e | x <- xs, y <- ys, z <- zs] ⇒
```

October 7, 2002 http://www.csg.lcs.mit.edu/6.827

L9-10
Arvind

A More General Solution

- Flatten the list after each map.
- Start the process by turning the expression into a one element list

```
[ e | x <- xs] ⇒
   concat (map (\x-> [e]) xs)

[ e | x <- xs, y <- ys] ⇒
   concat (map (\x->

[ e | x <- xs, y <- ys, z <- zs] ⇒
   concat (map (\x->
```

October 7, 2002 http://www.csg.lcs.mit.edu/6.827

Eliminate the intermediate list

```
[ e | x <- xs] ⇒ concat (map (\x-> [e]) xs)
```

Notice **map** creates a list which is immediately consumed by **concat** This intermediate list is avoided by **concatMap**

```
concatMap f []     = []
concatMap f (x:xs) = (f x) ++ (concatMap f xs)
```

```
[ e | x <- xs] ⇒ concatMap (\x-> [e]) xs

[ e | x <- xs, y <- ys] ⇒

      concatMap (\x->


[ e | x <- xs, y <- ys, z <- zs] ⇒

      concatMap (\x->
```


List Comprehensions with Predicates

```
[ e | x <- xs, p ] ⇒

   (map (\x-> e) (filter (\x-> p) xs)

   concatMap (\x-> if p then [e] else []) xs

[ e | x <- xs, p, y <- ys] ⇒

   concatMap (\x-> if p then
```

L9-13
Arvind

List Comprehensions:
First Functional Implementation- Wadler

```
TE[[[ e | x <- L, Q]]] =
          concatMap (\x-> TE[[[e | Q]]]) L

TE[[[ e | B, Q]]]      =
          if B then TE[[[e | Q]]] else []

TE[[[ e | ]]]          = e :[]
```

Can we avoid concatenation altogether?

L9-14
Arvind

Building the output from right-to-left

```
[ e | x <- xs, y <- ys] ⇒
  concat (map (\x-> map (\y-> e) ys) xs)
```

versus

```
[ e | x <- xs, y <- ys] ⇒
     let f []      = []
         f (x:xs’) =
               let  g []      = f xs’
                    g (y:ys’) = e:(g ys’)
               in
                    (g ys)
     in
         (f xs)
```

List Comprehensions:
Second Functional Implementation-Wadler

```
TE[[[ e | Q]]]   = TQ[[[e | Q] ++ []]]

TQ[[[ e | x <- L1, Q] ++ L ]] =
        let f []     = L
            f (x:xs) = TQ[[[ e | Q] ++ (f xs)]]
        in
            (f L1)

TQ[[[ e | B, Q] ++ L ]] =
             if B then TQ[[[ e | Q] ++ L ]]
                  else L

TQ[[ [ e | ] ++ L ]]     = e : L
```

This translation is efficient because it never flattens. The list is built right-to-left, consumed left-to-right.

The Correctness Issue

How do we decide if a translation is *correct?*

- if it produces the same answer as some reference translation, or
- if it obeys some other high-level laws

In the case of comprehensions one may want to prove that a translation satisfies the comprehension rewrite rules.

Pattern Matching

Desugaring Function Definitions

Function def ⇒ λ-expression + Case

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

⇒

```
map = (\t1 t2 ->
          case (t1,t2) of
            (f, [])      -> []
            (f,(x:xs)) -> (f x):(map f xs)
```

We compile the pattern matching using a tuple.

Complex to Simple Patterns

```
last []            =  e1
last [x]           =  e2
last (x1:(x2:xs))  =  e3
   ⇒
last = \t ->
    case t of
      []       -> e1
      (t1:t2) ->
```


Pattern Matching and Strictness

pH uses top-to-bottom, left-to-right order in pattern matching. This still does not specify if the pattern matching should force the evaluation of an expression

```
case (e1,e2) of
     ([]    , y) -> eb1
     ((x:xs), z) -> eb2
```

Should we valuate **e2**?
If not then the above expression is the same as

pH tries to evaluate minimum number of arguments.

Order of Evaluation and Strictness

Is there a minimum possible evaluation of an expression for pattern matching?

```
case (x,y,z) of              case (z,y,x) of
      (x,y,1) -> e1                (1,y,x) -> e1
      (1,y,0) -> e2    vs          (0,y,1) -> e2
      (0,1,0) -> e3                (0,1,0) -> e3
```

Very subtle differences - programmer should write *order-insensitive, disjoint* patterns.

Pattern Matching: *Syntax & Semantics*

Let us represent a case as **(*case* e *of* C)** where **C** is

$$C = P \to e \mid (P \to e) , C$$

$$P = x \mid CN_0 \mid CN_k(P_1, \ldots, P_k)$$

The rewriting rules for a case may be stated as follows:

(*case* e *of* P -> e1, C)

⇒ **e1** if match**(P,e)**

⇒ if ~match**(P,e)**

(*case* e *of* P -> e1)

⇒ **e1** if match**(P,e)**

⇒ if ~match**(P,e)**

The match Function

P = x | CN_0 | $CN_k(P_1, …, P_k)$

```
match[[x, t]]      = True

match[[CN0, t]]    = CN0 == tag(t)

match[[CNk(P1, …,Pk), t]] =
                   if tag(t) == CNk
                   then
                     (match[[P1, proj1(t)]] &&
                      .
                      .
                      .
                      match[[Pk, projk(t)]])
                   else
                      False
```


pH Pattern Matching

```
TE[[(case e of C)]]   =
     (let t = e in TC[[t, C]])

TC[[t, (P -> e)]]     =
     if match[[P, t]],
          then (let bind[[P, t]] in e)
          else error "match failure"

TC[[t, ((P -> e),C)]] =
     if match[[P, t]]
          then (let bind[[P, t]] in e)
          else TC[[t, C]]
```

Pattern Matching: bind Function

bind[[x, t]] = x = t

bind[[CN_0 , t]] = ε

bind[[$CN_k(P_1, \ldots, P_k)$, t]] =
bind[[P_1, $proj_1$(t)]] ;
.
.
.
bind[[P_k, $proj_k$(t)]]

Refutable vs Irrefutable Patterns

Patterns are used in binding for destructuring an expression---but what if a pattern fails to match?

```
let (x1, x2)       = e1
     x : xs        = e2
     y1: y2 : ys   = e3
in
   e
```

what if **e2** *evaluates to* **[]** *?*
e3 *to a one-element list ?*

Should we disallow refutable patterns in bindings?
Too inconvenient!

Turn each binding into a case expression