Compiling Pattern Matching and List Comprehensions

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 5, 2006

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

Pattern Matching

Desugaring Function Definitions

Function def $\Rightarrow$ λ-expression + Case

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

$\Rightarrow$

```
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) ->
           case t2 of
             []       -> let x  = t1
                          in e2
             (t3:t4) -> let x1 = t1
                             x2 = t3
                             xs = t4
                          in e3
```

Turn every case into a primitive case

Pattern Matching and Strictness

Haskell uses top-to-bottom, left-to-right order in pattern matching.

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

Strictness issue: *Should we evaluate* ***e2****?*

If not then the above expression is the same as

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

Order of Evaluation and Strictness

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

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

What about …

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

We must evaluate z

if z is 0 then we must evaluate x

if x is 0 then we must evaluate y

Is this what top-to-bottom, left-to-right pattern matching will do?

Be careful about ordering ....

Pattern Matching:

Abstract Syntax & Semantics

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

```
C = P -> e  | (P -> e) , C

P = x | CN_0 | CN_k(P_1, …,P_k)
```

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

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

$\Rightarrow$ **e1** if match**(P,e)**

$\Rightarrow$ **(*case* e *of* C)** if ~match**(P,e)**

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

$\Rightarrow$ **e1** if match**(P,e)**

$\Rightarrow$ **error** if ~match**(P,e)**

Is it top to bottom?

The match Function

```
P = x | CN0 | CNk(P1, …,Pk)
```

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

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

match[[CNk(P1, …,Pk), t]] =
      if tag(t) /= CNk
      then False
      else if not match[[P1, proj1(t)]]
           then False
           else ...
                if not match[[Pk, projk(t)]]
                then False
                else True
```

Is it left to right?

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

Compiling List Comprehensions

List Comprehensions: Abstract 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]` ⇒ `(`*`let`*` x = `$e_x$` `*`in`*` [e | Q]) ++ [e	x <- `$e_{xs}$`, Q]`		
Rule 2.1	`[e	False, Q]`	⇒	`[]`
Rule 2.2	`[e	True , Q]`	⇒	`[e	Q]`
Rule 3	`[e	]`	⇒	`e : []`

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?

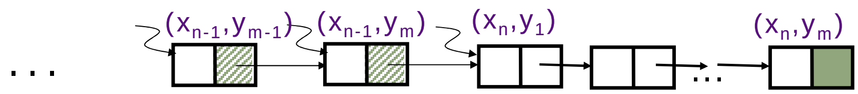
Idea: *Build the list from right-to-left*

TQ[[`[e | Q] ++ L`]]

where **L** has already been translated.

Building the output from right-to-left

[e | x <- xs, y <- ys]



Building the output from right-to-left

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

versus

```
[ e | x <- xs] ⇒
       let f []        = []
           f (x:xs')   = e : (f xs')
       in
           (f xs)
```

Similar but no need for concatenation

Building the output from right-to-left

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

Still hogging lot of stack

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.

More efficient and parallelizable translations in pH using I-Structures

I-Structures are write-once type of data structures

`The syntax shown here is for pH`

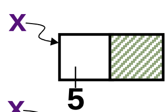
I-lists

```
data IList t = INil
             | ICons {hd ::t, tl:: .(IList t)}
```

I-Structure field

Allocation

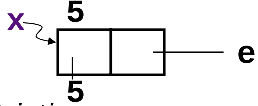
```
x = ICons {hd = 5}
```



a hole

Assignment

```
tl x := e
```



The single assignment restriction.
If violated the program will blow up.

Selection

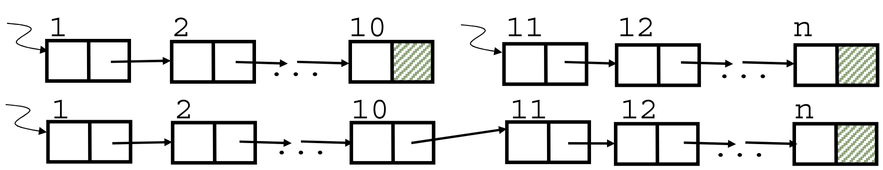
```
case xs of
    INil      -> ...
    ICons h t -> ...
```

we can also write `ICons {hd=h, tl=t} -> ...`

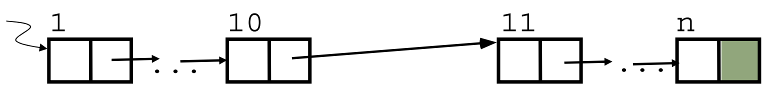
Open List Operations

A pair of I-list pointers for the *header* and the *trailer* cells.

joining two open lists



closing an open list



Open List Operation Definitions

```
type open_list t = ((IList t), (IList t))

nil_ol = (INil, INil)

close (hr,tr) =
       let
            case hr of
               INil -> ()
               ICons _ _ -> {tl tr := INil}
        in  cnv_Ilist_to_list hr

join (hr1,tr1) (hr2,tr2) =
        case hr1 of
            INil -> (hr2,tr2)
            ICons _ _ -> let tl tr1 := hr2
                         in (hr1,tr2)
```


Map Using Open Lists

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

- *Inefficient* because *it is not tail recursive!*
- A tail recursive version can be written using open lists:
  `map f xs = close (open_map f xs)`
  where

```
open_map f []     = (INil, INil)
open_map f (x:xs) =
      let tr   = ICons {hd=(f x)}
          last = for x' <- xs do
                    tr' = ICons {hd=(f x')}
                    tl tr := tr'
                    next tr = tr'
                 finally tr
      in (tr,last)
```

At each stage a side-effect to the "last" cell is caused.

Implementing List Comprehensions

Functional solution 1

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

- Inefficient even with tail recursive map because of too much *cons*ing

Functional solution 2

```
[ 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)
```

- Builds the list from right-to-left and avoids excessive *cons*ing but is sequential and hogs stack space

Implementing List Comprehensions Using Open Lists

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

1. Make n *open lists*, one for each x in xs
2. Join these lists together

```
let
     zs = nil_ol
in
     for x <- xs do
        z' = open_map (\y-> e) ys
        next zs = join zs z'
     finally zs
```

- *This solution eliminates all copying and preserves parallelism.*