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 using maps etc.
  - Pattern Matching: eliminate complex pattern matching using simple case-expressions
Pattern Matching

Desugaring Function Definitions

Function def $\Rightarrow$ $\lambda$-expression + Case

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

$\Rightarrow$

map = ($\lambda$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

\begin{align*}
\text{last} \; [\;] & = e_1 \\
\text{last} \; [x] & = e_2 \\
\text{last} \; (x_1:(x_2:xs)) & = e_3 \\
\Rightarrow \quad \text{last} & = \lambda t \rightarrow \\
& \quad \text{case } t \text{ of} \\
& \qquad [] \rightarrow e_1 \\
& \qquad (t_1:t_2) \rightarrow \\
& \quad \quad \text{case } t_2 \text{ of} \\
& \qquad \qquad [] \rightarrow \text{let } x = t_1 \\
& \qquad \qquad \quad \text{in } e_2 \\
& \qquad \qquad (t_3:t_4) \rightarrow \text{let } x_1 = t_1 \\
& \qquad \qquad \quad x_2 = t_3 \\
& \qquad \qquad \quad xs = t_4 \\
& \qquad \qquad \quad \text{in } e_3
\end{align*}

Pattern Matching and Strictness

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

\begin{center}
\begin{align*}
\text{case } (e_1,e_2) \text{ of} \\
& [(\;], \; y) \rightarrow eb_1 \\
& \quad ((x:xs), \; z) \rightarrow eb_2
\end{align*}
\end{center}

Strictness issue: \textit{Should we evaluate } e_2? \linebreak If not then the above expression is the same as

\begin{center}
\begin{align*}
\text{case } e_1 \text{ of} \\
& [] \rightarrow \text{let } y = e_2 \text{ in } eb_1 \\
& (x:xs) \rightarrow \text{let } z = e_2 \text{ in } eb_2
\end{align*}
\end{center}
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 \((\text{case } e \text{ of } C)\) where \(C\) is

\[
C = P \rightarrow e \mid (P \rightarrow e), C
\]

\[
P = x \mid CN_0 \mid CN_k(P_1, ..., P_k)
\]

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

\[\begin{align*}
(\text{case } e \text{ of } P \rightarrow e1, C) & \Rightarrow e1 & \text{if match } (P,e) \\
& \Rightarrow (\text{case } e \text{ of } C) & \text{if } \neg\text{match } (P,e) \\
(\text{case } e \text{ of } P \rightarrow e1) & \Rightarrow e1 & \text{if match } (P,e) \\
& \Rightarrow \text{error} & \text{if } \neg\text{match } (P,e)
\end{align*}\]
The match Function

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

\[
\begin{align*}
\text{match}[[x, t]] &= True \\
\text{match}[[CN_0, t]] &= CN_0 == \text{tag}(t) \\
\text{match}[[CN_k(P_1, \ldots, P_k), t]] &= \\
&\quad \text{if tag}(t) /= CN_k \\
&\quad \quad \text{then False} \\
&\quad \quad \quad \text{else if not match}[[P_1, \text{proj}_1(t)]] \\
&\quad \quad \quad \quad \text{then False} \\
&\quad \quad \quad \quad \quad \text{else ...} \\
&\quad \quad \quad \quad \quad \quad \text{if not match}[[P_k, \text{proj}_k(t)]] \\
&\quad \quad \quad \quad \quad \quad \quad \text{then False} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \text{else True}
\end{align*}
\]

Pattern Matching

\[
\begin{align*}
\text{TE}[[\text{case } e \text{ of } C]] &= (\text{let } t = e \text{ in } \text{TC}[[t, C]]) \\
\text{TC}[[t, (P \rightarrow e)]] &= \\
&\quad \text{if match}[[P, t]] \\
&\quad \quad \text{then (let bind}[[P, t]] \text{ in e) } \\
&\quad \quad \text{else error “match failure”}
\end{align*}
\]

\[
\text{TC}[[t, ((P \rightarrow e), C)]] = \\
&\quad \text{if match}[[P, t]] \\
&\quad \quad \text{then (let bind}[[P, t]] \text{ in e) } \\
&\quad \quad \text{else TC}[[t, C]]
\]


October 5, 2006  http://www.csg.csail.mit.edu/6.827  L09-10
Pattern Matching: bind Function

\[
\begin{align*}
\text{bind}[[x, t]] &= x = t \\
\text{bind}[[CN_0, t]] &= \varepsilon \\
\text{bind}[[CN_k(P_1, \ldots, P_k), t]] &= \\
&\quad \text{bind}[[P_1, proj_1(t)]] ; \\
&\quad \ldots \\
&\quad \text{bind}[[P_k, proj_k(t)]]
\end{align*}
\]

Refutable vs Irrefutable Patterns

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

\[
\begin{align*}
\text{let } (x1, x2) &= e1 \\
x : xs &= e2 \\
y1 : y2 : ys &= e3 \\
\text{in } \\
e
\end{align*}
\]

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 ] \Rightarrow []

Rule 1.2 \[ e | x <- (e_x : e_{xs}), Q ] \Rightarrow
(let x = e_x in [ e | Q ]) ++
[ e | x <- e_{xs}, Q ]

Rule 2.1 \[ e | \text{False}, Q ] \Rightarrow []

Rule 2.2 \[ e | \text{True}, Q ] \Rightarrow [ e | Q ]

Rule 3 \[ e | ] \Rightarrow e : []

List Comprehensions:
First Functional Implementation- Wadler

\[
\begin{align*}
\text{TE}[[( e | x <- L, Q)]] &= \text{concatMap } (\lambda x \rightarrow \text{TE}[[ e | Q]])L \\
\text{TE}[[ e | B, Q]] &= \begin{cases} 
\text{TE}[[ e | Q]] & \text{if } B \\
[ ] & \text{else}
\end{cases} \\
\text{TE}[[ e | ]]] &= e : []
\end{align*}
\]

Can we avoid concatenation altogether?

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

\[
\text{TQ}[[ e | Q] ++ L ]]
\]

where L has already been translated.
Building the output from right-to-left

\[ [e \mid x <- xs, y <- ys] \]

... 

\[ (x_{n-1}, y_{m-1}) \rightarrow (x_{n-1}, y_m) \rightarrow (x_n, y_1) \rightarrow (x_n, y_m) \rightarrow \ldots \]

Building the output from right-to-left

\[ [e \mid x <- xs] \Rightarrow \text{concatMap} (\lambda x \rightarrow [e]) \; xs \]

versus

\[ [e \mid x <- xs] \Rightarrow \]

\[ \text{let } f \; [] = [] \]

\[ f \; (x:xs') = e : (f \; xs') \]

\[ \text{in} \]

\[ (f \; xs) \]

Similar but no need for concatenation
Building the output from right-to-left

\[ [ e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \]
\[
\text{concatMap } (\lambda x\rightarrow \text{concatMap } (\lambda y\rightarrow e) \text{ ys}) \text{ xs}
\]

versus

\[ [ e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \]
\[
\text{let } f [] = []
\[
\text{f } (x:xs') =
\[
\text{let } g [] = f \text{ xs'}
\[
g (y:ys') = e :(g \text{ ys'})
\[
\text{in}
\[
(g \text{ ys})
\[
\text{in }
\[
(f \text{ xs})
\]

Still hogging lot of stack

List Comprehensions:
Second Functional Implementation-Wadler

\[
\text{TQ}[[[e \mid Q]]][[L]] =
\]
\[
\text{if } B \text{ then } \text{TQ}[[[e \mid Q]]][[L]] \text{ else } L
\]
\[
\text{TQ}[[[e \mid ]]][[L]] = e : L
\]

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

```hs
data ILList t = INil
               | ICons {hd :::t, tl::: (ILList t)}
```

**Allocation**

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

**Assignment**

```hs
tl x := e
```

*The single assignment restriction.*

If violated the program will blow up.

**Selection**

```hs
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

```hs```

*closing* an open list
Open List Operation Definitions

\[
\text{type open_list t = ((IList t), (IList t))}
\]

\[
\text{nil_ol = (INil, INil)}
\]

\[
\text{close (hr, tr) =}
\]

\[
\text{let}
\]

\[
\text{case hr of}
\]

\[
\text{INil \rightarrow ()}
\]

\[
\text{ICons \_ \_ \_ \_ \_ \rightarrow } \{\text{tl tr := INil}\}
\]

\[
\text{in cvn_Ilist_to_list hr}
\]

\[
\text{join (hr1, tr1) (hr2, tr2) =}
\]

\[
\text{case hr1 of}
\]

\[
\text{INil \rightarrow (hr2, tr2)}
\]

\[
\text{ICons \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \rightarrow } \text{let tl tr1 := hr2}
\]

\[
\text{in (hr1, tr2)}
\]

Map Using Open Lists

**Inefficient** because it is not tail recursive!

A tail recursive version can be written using open lists:

\[
\text{map f xs = close (open_map f xs)}
\]

where

\[
\text{open_map f [] = (INil, INil)}
\]

\[
\text{open_map f (x:xs) =}
\]

\[
\text{let tr = ICons \{hd = (f x)\}}
\]

\[
\text{last = for x' \leftarrow xs do}
\]

\[
\text{tr' = ICons \{hd = (f x')\}}
\]

\[
\text{tl tr := tr'}
\]

\[
\text{next tr = tr'}
\]

\[
\text{finally tr}
\]

\[
\text{in (tr, last)}
\]
Implementing List Comprehensions

Functional solution 1

\[ \{ e | x \leftarrow xs, y \leftarrow ys \} \Rightarrow \]
\[ \text{concatMap (} \lambda x \rightarrow \]
\[ \text{concatMap (} \lambda y \rightarrow [e] \) ys) xs \]

- Inefficient even with tail recursive map because of too much consing

Functional solution 2

\[ \{ e | x \leftarrow xs, y \leftarrow ys \} \Rightarrow \]
\[ \text{let } f \[] = [] \]
\[ f (x:xs') = \]
\[ \text{let } g \[] = f xs' \]
\[ g (y:ys') = e:(g ys') \]
\[ \text{in } (g ys) \]
\[ \text{in } (f xs) \]

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

Implementing List Comprehensions Using Open Lists

\[ \{ e | x \leftarrow xs, y \leftarrow ys \} \]

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

\[ \text{let } \]
\[ \quad \text{zs} = \text{nil}_\text{ol} \]
\[ \text{in} \]
\[ \quad \text{for } x \leftarrow xs \text{ do} \]
\[ \quad z' = \text{open_map (} \lambda y \rightarrow e \) ys \]
\[ \quad \text{next } zs = \text{join } zs z' \]
\[ \text{finally } zs \]

- This solution eliminates all copying and preserves parallelism.