Issues in Compiling Functional Languages

April 25, 2011
CS 132: Compiler Design
Pattern-Matching

- How do we compile the following function?

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```
Pattern-Matching

- The job of the *pattern compiler* is to turn complex pattern-matches like this into a sequence of "simple" tests.

```
zip xs' ys' =
  case xs' of
    []     -> []
    (x:xs) -> case ys' of
               []     -> []
               (y:ys) -> (x,y):(zip xs ys)
```
Pattern-Matching

• How about this function?

\[
\begin{align*}
  f \text{ 0 0} &= 0 \\
  f \text{ 0 1} &= 2 \\
  f \text{ _ _} &= 3
\end{align*}
\]
Pattern-Matching

• Naive solution

```hs
f x y =
  case x of
    0 -> case y of
      0 -> 0
      _ -> case x of
        0 -> case y of
          1 -> 2
          _ -> 3
        _ -> 3
    _ -> case x of
      0 -> case y of
        1 -> 2
        _ -> 3
      _ -> 3
      _ -> 3
```

f 0 0 = 0
f 0 1 = 2
f _ _ = 3
Pattern-Matching

- Re-using past tests:

\[
f x y = \begin{cases} 
 0 & \text{if } x = 0 \\
 0 & \text{if } y = 0 \\
 1 & \text{if } x = 1 \\
 _ & \text{if } y = _ \\
 _ & \text{if } y = _ \\
 _ & \text{if } y = _ \\
 \end{cases}
\]

\[
f y x = \begin{cases} 
 0 & \text{if } y = 0 \\
 0 & \text{if } x = 0 \\
 1 & \text{if } x = 1 \\
 _ & \text{if } x = _ \\
 _ & \text{if } x = _ \\
 _ & \text{if } x = _ \\
 \end{cases}
\]

- \( f 0 0 = 0 \)
- \( f 0 1 = 2 \)
- \( f _ _ _ = 3 \)
Pattern-Matching

• How about this function?

```haskell
nodups lst =
case lst of
  []    -> []
  [x]   -> [x]
  (x:y:zs) -> if x=y then
              nodups(y:zs)
            else
              x : nodups(y:zs)
```
Pattern-Matching

- Avoiding duplicate tests:

```haskell
nodups lst =
  case lst of
    []    -> []
    (x:tmp) ->
      case tmp of
        []    -> [x]
        y:zs   -> if x==y then
                    nodups(y:zs)
                    else
                    x : nodups(y:zs)
```
Pattern Compilation

- The job of the pattern compiler is essentially to construct a decision tree.
  - Guess what the complexity of optimal decision-tree construction is?
  - So heuristics are used.
Implementing a Pattern Compiler

- **Generalizations:**
  - Want to match $n$ values simultaneously
  - Each "arm" of the case has $n$ patterns, all of which must match the $n$ values.

- **Specializations:**
  - We are matching against variables $u_1, \ldots, u_n$
  - None of the patterns involve pairs.
  - We only handle matches against list patterns
    - Integers and other datatypes can be added easily later.
The Match Function

• The code we are looking for will be denoted by:

\[
\text{match}([u_1, \ldots, u_n],
[(\ldots,\ldots), e_1),
\ldots,
(\ldots,\ldots), e_m)],
\]

\[
e_{ow}
\]
The Match Function

- **Case:** One of the columns contains only variables.
  - Drop the column and replace the variables with the value being matched against.

```plaintext
match(  [ u1, u2,   u3   ],
    [([ f,  []],     ys    ], e1),
    [ f,    x:xs,  [] ], e2),
    [ g,    x:xs, y:ys ], e3])]
  e_{ow}) =

match(  [ u2,   u3   ],
    [([ [],     ys    ], e1[f→u1]),
    [ x:xs,  []    ], e2[f→u1]),
    [ x:xs, y:ys    ], e3[g→u1])],
  e_{ow})
```
The Match Function

- Case: One of the columns contains only data constructors.
  - Separate out the cases for each form

```haskell
match([ u2, u3 ],
   ((( [], ys ], e1),
     [ x:xs, [] ], e2),
     [ x:xs, y:ys ], e3)],
  e_ow))

= case u2 of
  [] -> match( [ u3 ],
    ([] ys ], e1),
    e_ow)
  u4:u5 -> match ([ u4, u5, u3 ],
    ((( x, xs, [] ], e2),
      ([ x, xs, y:ys ], e3)],
    e_ow))
```
The Match Function

- Case: No more patterns to match against
  - Pick the first available arm

\[
\text{match}(\; [],
\quad ([[]], e_1),
\quad ([[]], e_2)\],
\quad e_{ow}) = e_1
\]

\[
\text{match}([],
\quad [],
\quad e_{ow}) = e_{ow}
\]
Example: zip

match([u1, u2],
    [[[[], ys], []],
     ([x:xs, []], []),
     ([x:xs, y:ys], (x,y):(zip xs ys))],
    ERROR)

case u1 of
    []    -> match([u2], [[ys], []], ERROR)
    u3:u4 -> match([u3, u4, u2],
                      [[[x, xs], []], []),
                      ([x, xs, y:ys], (x,y):(zip xs ys))],
                      ERROR)
Example: zip

```haskell
case u1 of
    []   -> match([], [[], []], ERROR)
    u3:u4 -> match([ u3, u4, u2 ],
                   [[[ x, xs, [] ]], []],
                   ([[ x, xs, y:ys ]], (x,y):(zip xs ys))],
                   ERROR)
```

```haskell
case u1 of
    []   -> match([], [[], []], ERROR)
    u3:u4 -> match([ u2 ],
                   [[[ [] ]], []],
                   ([ y:ys ], (u3,y):(zip u4 ys))],
                   ERROR)
```
Example: zip

case u1 of
  [] -> match([], [[[], []]], ERROR)
  u3:u4 -> match([ u2 ],
                     [[ [] ], []],
                     ([ y:ys ], (u3,y):(zip u4 ys))],
                 ERROR)

case u1 of
  [] -> []
  u3:u4 ->
    case u2 of
      [] -> []
      u5:u6 -> match([ u5, u6 ],
                      [[[ y, ys ], (u3,y):(zip u4 ys)]]],
                          ERROR))
Example: zip

case u1 of
    []   -> []
    u3:u4 ->
        case u2 of
            []   -> []
            u5:u6 -> match(  [ u5, u6 ],
                              [[[ y, ys ], (u3,y):(zip u4 ys)]],
                              ERROR))

case u1 of
    []   -> []
    u3:u4 ->
        case u2 of
            []   -> []
            u5:u6 -> (u3,u5):(zip u4 u6))
The Match Function

- Case: A column has both variables and constructors
  - Group successive cases together if they all have a variable in this column or they all have a constructor.
  - No reordering allowed

\[
\text{match}( \ [ u2, u3 \ ] , \\
[ [( [], [] ], e_0) , \\
[ [], ys ], e_1) , \\
[ xs, [] ], e_2) , \\
[ x:xs, y:ys ], e_3) ] , \\
e_{ow}) = \text{match}( \ [ u2, u3 ] , \\
[ [( [], [] ], e_0) , \\
[ [], ys ], e_1) ] , \\
\text{match}( \ [ u2, u3 ] , \\
[ [( xs, [] ], e_2) ] , \\
\text{match}( [ u2, u3 ] , \\
[ [ x:xs, y:ys ], e_3) ] , \\
e_{ow}))))
\]
Pattern Compilation Issues

• Which column to match on first?
  – Result may matter in languages where pattern-matching can cause side-effects.
    • If so, language may require left-to-right testing.
    – What about Haskell?

• How to avoid duplication of tests?
• How to avoid duplication of code?
First-Class Functions

• How can we compile functions that construct functions at run-time?
  – Code pointers as in C are not enough, because of free variables.

```make_adder (n::Int) =
  \(m::\text{Int}\) -> m+n

successor   = make_adder 1
predecessor = make_adder ( -1 )```
• One method of returning a function would be to invoke a compiler to create a specialized version of the code, and then return a pointer to this code.
• Extremely high overhead.

```javascript
make_adder (n:int) =
  (fn (m:int) -> m+n)

successor : int->int =
  make_adder 1
```
Closure Conversion

• Idea:
  – If functions had no free variables, then they could all be defined at top-level (as in C).
  – We can get rid of free variables by adding making them into an argument of the function (called the environment).
First Thought

\[
\begin{align*}
\text{let } & \ x = 1 \\
& \ y = 2 \\
& \ z = 3 \\
& \ f(w) = x + y + w \\
\text{in} & \\
& \ f \ 100
\end{align*}
\]

\[
\begin{align*}
\text{let } & \ x = 1 \\
& \ y = 2 \\
& \ z = 3 \\
& \ f((x_f, y_f), w) = x_f + y_f + w \\
\text{in} & \\
& \ f \ ((x, y), 100)
\end{align*}
\]
Intuition

```
let x = 1
    y = 2
    z = 3
    f(w) = x+y+w
in
    f 100
```

```
let x = 1
    y = 2
    z = 3
    f_{\text{code}}((x_f, y_f), w) = x_f+y_f+w
    f_{\text{env}} = (x, y)
f = (f_{\text{code}}, f_{\text{env}})
in
    (fst f)(snd f, 100)
```
Example 2

```plaintext
let x = 1
    y = 2
    z = 3
    f(w) = x + y + w
    g(q) = x + z + q
in
    (f 100) + (g 99)
end
```

```plaintext
let x = 1
    y = 2
    z = 3
    f_{code}((x_f,y_f),w) = x_f + y_f + w
    f_{env} = (x,y)
    f = (f_{code}, f_{env})
    g_{code}((x_g,z_g),q) = x_g + z_g + q
    g_{env} = (x,z)
    g = (g_{code}, g_{env})
in
    (fst f)(snd f, 100) + (fst g)(snd g, 99)
end
```
Variation: Shared Environments

```plaintext
let x = 1
  y = 2
  z = 3
  \( f_{\text{code}}((x_{\text{fg}}, y_{\text{fg}}, _), w) = x_{\text{fg}} + y_{\text{fg}} + w \)
  \( f_{\text{env}} = (x, y, z) \)
  \( f = (f_{\text{code}}, f_{\text{env}}) \)
  \( g_{\text{code}}((x_{\text{fg}}, _, z_{\text{fg}}), q) = x_{\text{fg}} + z_{\text{fg}} + q \)
  \( g = (g_{\text{code}}, f_{\text{env}}) \)
end

\((\text{fst } f)(\text{snd } f, 100) + (\text{fst } g)(\text{snd } g, 99)\)```
Example 3

make_adder(n)= \m -> n+m

make_adder_{\text{code}}(((), n) =
    \text{let } \text{anon}_{\text{code}}(n_a, m) = n_a + m \\
    \text{anon}_{\text{env}} = n \\
    \text{in} \\
    (\text{anon}_{\text{code}}, \text{anon}_{\text{env}})

make_adder_{\text{env}} = ()

make_adder = (make_adder_{\text{code}}, make_adder_{\text{env}})
Example 3 with Hoisting

\[
anon_{\text{code}}(n_a, m) = n_a + m
\]

\[
\text{make_adder}_{\text{code}}((), n) = \\
\quad \text{let } anon_{\text{env}} = n \\
\quad \text{in } (anon_{\text{code}}, anon_{\text{env}})
\]

\[
\text{make_adder}_{\text{env}} = ()
\]

\[
\text{make_adder} = (\text{make_adder}_{\text{code}}, \text{make_adder}_{\text{env}})
\]
Example 4 (Partial)

\[
f(z) = \text{let } g(x) = z + x + b + c \\
\text{in } g(a + b + c)
\]

\[
f_{\text{code}}((a', b', c'), z) \rightarrow \\
\text{let} \\
\quad g(x) = z + x + b' + c' \\
\quad \text{in} \\
\quad g(a' + b' + c') \\
\text{end} \\
f_{\text{env}} = (a, b, c) \\
f = (f_{\text{code}}, f_{\text{env}})
\]
Example 4: Flat Closures

\[ f(z) = \text{let } g(x) = z + x + b + c \text{ in } g(a + b + c) \]

\[ f_{\text{code}}((a_f, b_f, c_f), z) = \]
\[ \text{let } \]
\[ g_{\text{code}}((z_g, b_g, c_g), x) = z_g + x + b_g + c_g \]
\[ g_{\text{env}} = (z, b_f, c_f) \]
\[ g = (g_{\text{code}}, g_{\text{env}}) \]
\[ \text{in } \]
\[ (\text{fst } g)(\text{snd } g, a_f + b_f + c_f) \]

\[ f_{\text{env}} = (a, b, c) \]
\[ f = (f_{\text{code}}, f_{\text{env}}) \]
Flat Closures with Hoisting

\[ g_{\text{code}}((z_g, b_g, c_g), x) = z_g + x + b_g + c_g \]

\[ f_{\text{code}}((a_f, b_f, c_f), z) = \]
\[
\begin{align*}
\text{let } & g_{\text{env}} = (z, b_f, c_f) \\
& g = (g_{\text{code}}, g_{\text{env}}) \\
\text{in } & (\text{fst } g)(\text{snd } g, a_f + b_f + c_f)
\end{align*}
\]

\[ f_{\text{env}} = (a, b, c) \]

\[ f = (f_{\text{code}}, f_{\text{env}}) \]
Example 4: Linked Closures

\[ f(z) = \text{let } g(x) = z + x + b + c \]
\[ \text{in } g(a + b + c) \]

\[
g_{\text{code}}((f_{\text{env}}, z_g), x) =
\begin{align*}
\text{let } (_, b_g, c_g) &= f_{\text{env}} \\
\text{in } z_g + x + b_g + c_g
\end{align*}
\]

\[
f_{\text{code}}((a_f, b_f, c_f) @ env_f, z) =
\begin{align*}
\text{let } g_{\text{env}} &= (env_f, z) \\
g &= (g_{\text{code}}, g_{\text{env}}) \\
\text{in } (\text{fst } g)(\text{snd } g, a_f + b_f + c_f)
\end{align*}
\]

\[ f_{\text{env}} = (a, b, c) \]

\[ f = (f_{\text{code}}, f_{\text{env}}) \]
Example 5

```
let y = 1
in
  if True then
    \x -> x+y
  else
    \z -> z

let y = 1
in
  if True then
    ( \ (y_a::Int, x::Int) -> x+y_a), y )
  else
    ( \ (()), z::Int) -> z , () )
```
Closure-Conversion Summary

• Doing closure-conversion well is still an active topic of research
  – When is it safe to reuse environments?
  – Do closures have to contain *all* of the free variables?
  – When can we avoid constructing closures altogether?
Tailcalls

• Also known as "tail recursion".
• Summary:
  – Particularly efficient code possible when the last thing a function does is call another function.
  – Every real compiler for functional languages implements this optimization.

\[ f(x) = \text{if } (x>0) \text{ then } g(x) \text{ else } h(x) \]

\[ f(x) = \text{if } (x>0) \text{ then } f(f(x-1)) \text{ else } g(x):h(x) \]
Calls

- Naive implementation of a tailcall from f to g:
  - Save f's return address
  - Put the parameters in registers
  - Subroutine call to g
  - The function g returns back to f.
  - Restore f's return address
  - Pop f's stack frame
  - Return from f.
Tailcalls

• Optimized implementation of a tailcall from \textit{f} to \textit{g}:
  – Put the parameters for \textit{g} in registers
  – Put \textit{f}'s return address where \textit{g} expects to find its return address
  – Restore callee-save registers, if any.
  – Pop \textit{f}'s stack frame
  – One-way jump to \textit{g}.

• Advantages:
  – Early deallocation of \textit{f}'s stack space.
  – The function \textit{g} returns directly to \textit{f}'s caller.