Issues in Compiling Functional Languages

April 25, 2011

CS 132: Compiler Design

Pattern-Matching

- How do we compile the following function?

```
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?

```
f 0 0 = 0
f 0 1 = 2
f _ _ = 3
```

Pattern-Matching

```
f 0 0 = 0
f 0 1 = 2
f _ _ = 3
```

- Naive solution

```
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
```

Pattern-Matching

```
f 0 0 = 0
f 0 1 = 2
f _ _ = 3
```

- Re-using past tests:

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

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

Pattern-Matching

- How about this function?

```
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:

```
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:

$$
\begin{array}{l}
\mathbf{match}([u_1,\ldots,u_n], \\
\qquad [([pat_1^1,\ldots,pat_1^n],\ e_1), \\
\qquad\ \ldots, \\
\qquad\ ([pat_m^1,\ldots,pat_m^n],\ e_m)], \\
\qquad e_{ow})
\end{array}
$$

The Match Function

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

```
match(  [ u1, u2,   u3   ],
      [([ f,  [],   ys   ], e_1),
        [ f,  x:xs, []   ], e_2),
        [ g,  x:xs, y:ys ], e_3)],
      e_ow) =

match(  [ u2,   u3   ],
      [([ [],   ys   ], e_1[f→u1]),
        [ x:xs, []   ], e_2[f→u1]),
        [ x:xs, y:ys ], e_3[g→u1])],
      e_ow)
```

The Match Function

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

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

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

The Match Function

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

```
match(  [],
      [([], e_1),
       ([], e_2)],
      e_ow) = e_1

match([],
      [],
      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**

```
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)
```

```
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

```
match(  [ u2,   u3   ],
      [([ [],   []   ], e0),
        [ [],   ys   ], e1),
        [ xs,   []   ], e2),
        [ x:xs, y:ys ], e3)],
      eow)
 = match(   [u2, u3 ],
         [([ [], [] ], e0),
           [ [], ys ], e1)],
         match(  [ u2, u3 ]
               [([ xs, [] ], e2)],
               match([ u2,   u3   ],
                     [ x:xs, y:ys ], e3)],
                     eow)))
```

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::Int) -> m+n

successor   = make_adder 1
predecessor = make_adder ( -1 )
```

RTCG

- 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.

```
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

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

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

Intuition

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

```
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)
in
    (fst f)(snd f, 100)
```

Example 2

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

$$
\begin{array}{l}
\texttt{let x = 1} \\
\qquad \texttt{y = 2} \\
\qquad \texttt{z = 3} \\
\qquad \texttt{f}_{\texttt{code}}\texttt{((x}_{\texttt{f}}\texttt{,y}_{\texttt{f}}\texttt{),w) = x}_{\texttt{f}}\texttt{ + y}_{\texttt{f}}\texttt{ + w} \\
\qquad \texttt{f}_{\texttt{env}}\texttt{ = (x,y)} \\
\qquad \texttt{f \ \ = (f}_{\texttt{code}}\texttt{, f}_{\texttt{env}}\texttt{)} \\
\qquad \texttt{g}_{\texttt{code}}\texttt{((x}_{\texttt{g}}\texttt{,z}_{\texttt{g}}\texttt{),q) = x}_{\texttt{g}}\texttt{ + z}_{\texttt{g}}\texttt{ + q} \\
\qquad \texttt{g}_{\texttt{env}}\texttt{ = (x,z)} \\
\qquad \texttt{g \ \ = (g}_{\texttt{code}}\texttt{, g}_{\texttt{env}}\texttt{)} \\
\texttt{in} \\
\qquad \texttt{(fst f)(snd f, 100) + (fst g)(snd g, 99)} \\
\texttt{end}
\end{array}
$$

Variation: Shared Environments

```
let x = 1
    y = 2
    z = 3
    f_code((x_fg,y_fg,_),w) = x_fg + y_fg + w
    fg_env = (x,y,z)
    f   = (f_code, fg_env)
    g_code((x_fg,_,z_fg),q) = x_fg + z_fg + q
    g   = (g_code, fg_env)
in
    (fst f)(snd f, 100) + (fst g)(snd g, 99)
end
```

Example 3

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

$$
\begin{array}{l}
\texttt{make\_adder}_{\texttt{code}}\texttt{((), n) =} \\
\qquad \texttt{let anon}_{\texttt{code}}\texttt{(n}_{\texttt{a}}\texttt{, m) = n}_{\texttt{a}}\texttt{ + m} \\
\qquad \qquad \texttt{anon}_{\texttt{env}}\texttt{ = n} \\
\qquad \texttt{in} \\
\qquad \quad \texttt{(anon}_{\texttt{code}}\texttt{, anon}_{\texttt{env}}\texttt{)} \\
\\
\texttt{make\_adder}_{\texttt{env}}\texttt{ = ()} \\
\\
\texttt{make\_adder = (make\_adder}_{\texttt{code}}\texttt{, make\_adder}_{\texttt{env}}\texttt{)}
\end{array}
$$

Example 3 with Hoisting

```
anon_code(n_a, m) = n_a + m

make_adder_code((), n) =
       let anon_env = n
       in (anon_code, anon_env)

make_adder_env = ()

make_adder = (make_adder_code, make_adder_env)
```

Example 4 (Partial)

```
f(z) = let g(x) = z+x+b+c
       in g(a+b+c)
```

```
f_code((a',b',c'),z) ->
           let
              g(x) = z+x+b'+c'
           in
              g(a'+b'+c')
           end
f_env = (a,b,c)
f     = (f_code, f_env)
```

Example 4: Flat Closures

```
f(z) = let g(x) = z+x+b+c
       in g(a+b+c)
```

$$
\begin{array}{l}
\mathtt{f_{code}((a_f,b_f,c_f),z)\ =} \\
\qquad\qquad \mathtt{let} \\
\qquad\qquad\quad \mathtt{g_{code}((z_g,b_g,c_g),x)\ =\ z_g+x+b_g+c_g} \\
\qquad\qquad\quad \mathtt{g_{env}\ =\ (z,\ b_f,\ c_f)} \\
\qquad\qquad\quad \mathtt{g\ =\ (g_{code},\ g_{env})} \\
\qquad\qquad \mathtt{in} \\
\qquad\qquad\quad \mathtt{(fst\ g)(snd\ g,\ a_f+b_f+c_f)} \\
\\
\mathtt{f_{env}\ =\ (a,b,c)} \\
\\
\mathtt{f\quad\ =\ (f_{code},\ f_{env})}
\end{array}
$$

Flat Closures with Hoisting

```
g_code((z_g,b_g,c_g),x) = z_g+x+b_g+c_g


f_code((a_f,b_f,c_f),z) =
          let g_env = (z, b_f, c_f)
              g = (g_code, g_env)
          in
              (fst g)(snd g, a_f+b_f+c_f)

f_env = (a,b,c)

f   = (f_code, f_env)
```

Example 4: Linked Closures

```
f(z) = let g(x) = z+x+b+c
          in g(a+b+c)
```

```
g_code((fenv,z_g),x) =
                    let (_, b_g, c_g) = fenv
                    in z_g+x+b_g+c_g

f_code( (a_f,b_f,c_f) @ env_f, z) =
             let g_env = (env_f, z)
                 g     = (g_code, g_env)
             in
                 (fst g)(snd g, a_f+b_f+c_f)

f_env = (a,b,c)

f     = (f_code, f_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) = if (x>0) then g(x) else h(x)
```

```
f(x) = if (x>0) then f(f(x-1)) 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 **`f`** to **`g`**:
  - Put the parameters for **`g`** in registers
  - Put **`f`**'s return address where **`g`** expects to find its return address
  - Restore callee-save registers, if any.
  - Pop **`f`**'s stack frame
  - One-way jump to **`g.`**
- Advantages:
  - Early deallocation of **`f`**'s stack space.
  - The function **`g`** returns directly to **`f`**'s caller.