Losing Functions Without Gaining Data

Neil Mitchell, Colin Runciman
University of York
community.haskell.org/~ndm/firstify
The Goal

• Remove functional values
  – Only named functions defined at the top level
  – No under/over application

• Without introducing data
  – Don’t want to introduce new data values
  – Avoid encoding functions in data
The Purpose

- Analysis!
  - Termination checking
  - Strictness analysis
  - Pattern-match safety (eg. Catch, Haskell08)
Example 1

**sum :: [Int] → Int**

\[
\text{sum } xs = \text{foldl } (\lambda x y \rightarrow x + y) \ 0 \ xs
\]

**foldl :: (a → b → a) → a → [b] → a**

\[
\text{foldl } f \ z \ [] = z
\]

\[
\text{foldl } f \ z \ (x:xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]
Example 1: Result

\[
\begin{align*}
\text{sum} & \:: \ [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \ xs & = \text{foldl}^+_+ \ 0 \ xs
\end{align*}
\]

\[
\begin{align*}
\text{foldl}^+_+ & \:: \ a \rightarrow \ [b] \rightarrow \ a \\
\text{foldl}^+_+ \ z \ [\ ] & = z \\
\text{foldl}^+_+ \ z \ (x:xs) & = \text{foldl}^+_+ \ (z + x) \ xs
\end{align*}
\]

Ingredient: specialisation
Example 2

apply :: String → Int → Int
apply str x = case meaning str of
    Just f → f x
    Nothing → x

meaning :: String → Maybe (Int → Int)
meaning "abs" = Just abs
meaning _ = Nothing
Example 2: Result

apply :: String → Int → Int
apply str x = case str of
  "abs" → abs x
  _ → x

Ingredients: inlining, simplification
The Central Idea

- Introduce explicit lambdas
  - Makes higher-order bits easier to see
- Move the lambdas around
  - The bulk of the work
- Eliminate lambdas
  - Applied lambdas
  - Unused lambdas
Moving Lambdas Around

- Inlining
- Simplification
- Specialisation

+ Restrictions

First-order reduction
Purpose of Each Stage

- **Simplification**
  - Eliminate applied lambdas

- **Inlining**
  - Eliminate functions returning lambdas inside constructors

- **Specialisation**
  - Eliminate lambdas passed as arguments
Simplification

- Lots of basic simplifications
  - eg. case/case, case of known constructor, application of a lambda

- Also need these let rules
  - let \( v = x \) in \( \lambda w \rightarrow y \) \( \Rightarrow \) \( \lambda w \rightarrow \) let \( v = x \) in \( y \)
  - let \( v = x \) in \( y \) \( \Rightarrow \) \( y [x / v] \),
    if \( x \) is a lambda or a boxed lambda
Boxed Lambda

- Syntactic condition, under-approximates...
- ...expressions whose results are constructions with a lambda component

<table>
<thead>
<tr>
<th>Boxed Lambda’s</th>
<th>Not Boxed Lambda’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\lambda x \rightarrow x])</td>
<td>(\lambda x \rightarrow [x])</td>
</tr>
<tr>
<td>Just ([\lambda x \rightarrow x])</td>
<td>[foo ((\lambda x \rightarrow x))]</td>
</tr>
<tr>
<td>let y = 1 in ([\lambda x \rightarrow x])</td>
<td>foo ([\lambda x \rightarrow x])</td>
</tr>
<tr>
<td>[Nothing, Just ((\lambda x \rightarrow x))]</td>
<td>let v = ([\lambda x \rightarrow x]) in v</td>
</tr>
</tbody>
</table>
Inlining

• Purpose: eliminate functions returning boxed lambdas

• case f xs of ... ⇒ case {body f} xs of ...
  – where {body f} is boxed lambda
Specialisation

- Purpose: eliminate lambdas passed to functions
- Given \( f \ e_1 \ldots e_n \), where some \( e_i \) is a lambda or boxed lambda
- Produce specialised \( f' \)
  - eliminate the \( i^{th} \) argument
  - introduce argument for each free variable in \( e_i \)
- Reformulate the application to use \( f' \)
Specialisation Example

1. \( \text{sum } xs = \text{foldl} (\lambda x \ y \rightarrow x + y) \ 0 \ xs \)

2. \( \text{foldl}_+ z \ xs = \text{foldl} (\lambda x \ y \rightarrow x + y) \ z \ xs \)

3. \( \text{sum } xs = \text{foldl}_+ 0 \ xs \)
Where the Lambdas Go

• Functions returning lambdas are eta expanded
• Functions returning boxed lambdas are inlined
• Functions with lambda arguments are specialised
• All other lambdas are targets for simplification rules

No lambda can hide!
Termination

- Specialisation may not terminate
  - Limited by homeomorphic embedding
- Inlining may not terminate
  - Limited by local numeric bounds
- Limits force termination when lambdas used to store an unbounded amount of information (eg. difference lists)
Disclaimers

• Not complete: may be residual lambdas if
  – Termination criteria kick in
  – Lambdas are passed to primitive functions
  – Root function takes/returns lambdas

• Loss of sharing

\[
f \ x = \text{let } i = \text{expensive } x \text{ in } \lambda j \rightarrow i + j
\]

\[
\Rightarrow
\]

\[
f \ x = \lambda j \rightarrow \text{let } i = \text{expensive } x \text{ in } i + j
\]
Results

• Successful on 62 of 66 nofib programs
  – Not cacheprof, grep, lift, prolog
• ~0.5 seconds to transform a program
  – Best = 0.1, Worst = 1.2
• Average code-size reduction of 30%
  – Best = 78% reduction, Worst = 27% increase
• Catch (Haskell08) relies on this method
  – 3 real bugs in HsColour
Results: Strictness

• Ask GHC – is add’s second arg strict?

add :: Int → Int → Int
add x y = apply 10 (+x) y

apply :: Int → (a → a) → a → a
apply 0 f x = x
apply n f x = apply (n - 1) f (f x)
Results: Termination

• Ask Agda – does this terminate?

```haskell
cons : (N → List N) → N → List N
cons f x = x :: f x

downFrom : N → List N
downFrom = cons f
where f : N → List N
  f zero = [ ]
  f (suc x) = downFrom x
```
Conclusions

• Let’s analyse higher-order programs!

• Write first-order analysis pass
• Old way: extend to higher-order
  – ~5 years for strictness analysis
• New way: use defunctionalisation
  – ~0.5 seconds