Binders Unbound

Stephanie Weirich[1] Brent Yorgey[1] Tim Sheard[2]

[1]University of Pennsylvania

[2]Portland State University

NJPLS
Princeton University
April 8, 2011

Lambda calculus

Let's implement the lambda calculus:

$$t ::= x \mid t\ t \mid \lambda x.t$$

in Haskell.

Lambda calculus

Let's implement the lambda calculus:

$$t ::= x \mid t\ t \mid \lambda x.t$$

in Haskell.

First try:

$$\begin{aligned}\textbf{data}\ E = {} & \mathit{Var}\ \mathit{String} \\ \mid {} & \mathit{App}\ E\ E \\ \mid {} & \mathit{Lam}\ \mathit{String}\ E\end{aligned}$$

Lambda calculus

Let's implement the lambda calculus:

$$t ::= x \mid t\ t \mid \lambda x.t$$

in Haskell.

First try:

$$\begin{aligned} \textbf{data}\ E = &\ \mathit{Var}\ \mathit{String} \\ \mid &\ \mathit{App}\ E\ E \\ \mid &\ \mathit{Lam}\ \mathit{String}\ E \end{aligned}$$

$$\begin{aligned} &\mathsf{subst} :: \mathit{String} \to E \to E \to E \\ &\mathsf{subst}\ x\ u\ t = \ldots? \end{aligned}$$

...with UNBOUND

$$t ::= x \mid t\ t \mid \lambda x.t$$

$$\begin{aligned}
&\textbf{type}\ N = \textsf{\textbf{Name}}\ E\\
&\textbf{data}\ E = \mathit{Var}\ N\\
&\qquad\quad \mid \mathit{App}\ E\ E\\
&\qquad\quad \mid \mathit{Lam}\ (\textsf{\textbf{Bind}}\ N\ E)
\end{aligned}$$

. . . with UNBOUND

$$t ::= x \mid t\ t \mid \lambda x.t$$

$$\begin{aligned}
&\textbf{type } N = \textsf{\textbf{Name}}\ E \\
&\textbf{data } E = \mathit{Var}\ N \\
&\qquad\qquad \mid \mathit{App}\ E\ E \\
&\qquad\qquad \mid \mathit{Lam}\ (\textsf{\textbf{Bind}}\ N\ E)
\end{aligned}$$

. . . and now we get these for free!

$$\begin{aligned}
\mathsf{subst} &:: N \to E \to E \to E \\
\mathsf{fv} &:: E \to [N] \\
\ldots
\end{aligned}$$

Example (parallel reduction)

$$
\begin{array}{l}
\mathit{red} :: \mathsf{Fresh}\ m \Rightarrow E \rightarrow m\ E \\
\mathit{red}\ (\mathit{Var}\ x) = \mathit{return}\ (\mathit{Var}\ x) \\
\mathit{red}\ (\mathit{Lam}\ b) = \mathbf{do} \\
\quad (x, e) \leftarrow \mathsf{unbind}\ b \\
\quad e' \qquad \leftarrow \mathit{red}\ e \\
\quad \mathbf{case}\ e'\ \mathbf{of} \\
\qquad \mathit{App}\ e''\ (\mathit{Var}\ y) \\
\qquad\quad \mid x \equiv y \wedge \neg\ (x \in \mathsf{fv}\ e'') \\
\qquad\qquad \rightarrow \mathit{return}\ e'' \\
\qquad \_ \qquad\ \ \rightarrow \mathit{return}\ (\mathit{Lam}\ (\mathsf{bind}\ x\ e'))
\end{array}
$$

Example (parallel reduction)

$$
\begin{array}{l}
\mathit{red}\ (\mathit{App}\ e_1\ e_2) = \mathbf{do} \\
\quad e_1' \leftarrow \mathit{red}\ e_1 \\
\quad \mathbf{case}\ e_1'\ \mathbf{of} \\
\quad\quad \mathit{Lam}\ b \rightarrow \mathbf{do} \\
\quad\quad\quad (x, e') \leftarrow \mathsf{unbind}\ b \\
\quad\quad\quad e_2' \qquad \leftarrow \mathit{red}\ e_2 \\
\quad\quad\quad \mathit{return}\ (\mathsf{subst}\ x\ e_2'\ e') \\
\quad\quad \_ \rightarrow \mathbf{do} \\
\quad\quad\quad e_2' \leftarrow \mathit{red}\ e_2 \\
\quad\quad\quad \mathit{return}\ (\mathit{App}\ e_1'\ e_2')
\end{array}
$$

UNBOUND provides a set of <u>type combinators</u> for expressing binding structure.

What other sorts of binding structure can we encode?

Binding multiple names

Instead of $\lambda x.\ \lambda y.\ \lambda z.\ t$,

$$\lambda\ x\ y\ z.\ t$$

Binding multiple names

Instead of $\lambda x.\ \lambda y.\ \lambda z.\ t$,

$$\lambda\ x\ y\ z.\ t$$

$$\begin{aligned}\textbf{data}\ E = \ & ...\\ \mid\ & Lam\ (\textsf{\textbf{Bind}}\ [N]\ E)\end{aligned}$$

Let

let $x = e_1$ in e_2

(x bound in e_2)

Let

$$\text{let } x = e_1 \text{ in } e_2$$

$$(x \text{ bound in } e_2)$$

First try:

$$\begin{aligned} \mathbf{data}\ E = \ & \ldots \\ & \mid\ \mathit{Let}\ E\ (\textbf{Bind}\ N\ E) \end{aligned}$$

Let

$$\text{let } x = e_1 \text{ in } e_2$$

$$(x \text{ bound in } e_2)$$

Better:

$$\begin{aligned} \mathbf{data}\ E = \ & ... \\ & \mid\ Let\ (\textsf{\textbf{Bind}}\ (N, \textsf{\textbf{Embed}}\ E)\ E) \end{aligned}$$

Multi-let

$$\text{let } x_1 = e_1, \ \ldots, \ x_n = e_n \text{ in } e$$

$$(x_i \text{ bound in } e)$$

$$\begin{aligned} \mathbf{data}\ E = \ & ... \\ \mid \ & \mathit{Let}\ (\textbf{Bind}\ [(N, \textbf{Embed}\ E)]\ E) \end{aligned}$$

Recursive binding

How about

$$\text{letrec } x_1 = e_1,\ \ldots,\ x_n = e_n \text{ in } e$$

$$(x_i \text{ bound in } e \text{ and all } e_j)?$$

Recursive binding

How about

$$\text{letrec } x_1 = e_1,\ \ldots,\ x_n = e_n \text{ in } e$$

$$(x_i \text{ bound in } e \text{ and all } e_j)?$$

Recursive binding:

$$\begin{aligned} \mathbf{data}\ E = \ & \ldots \\ & \mid\ \mathit{Letrec}\ (\textbf{Bind}\ (\textbf{Rec}\ [(N, \textbf{Embed}\ E)])\ E) \end{aligned}$$

let*

What about

$$\text{let* } x_1 = e_1,\ \ldots,\ x_n = e_n \text{ in } e$$

$$(x_i \text{ bound in } e \text{ and } e_j \text{ for } j > i)?$$

let*

What about

$$\text{let* } x_1 = e_1,\ \ldots,\ x_n = e_n \text{ in } e$$

$$(x_i \text{ bound in } e \text{ and } e_j \text{ for } j > i)?$$

Working but suboptimal:

$$\begin{aligned}
\mathbf{data}\ \mathit{LetList} &= \mathit{Body}\ E \\
&\mid\ \mathit{Binding}\ (\textsf{\textbf{Bind}}\ (N, \textsf{\textbf{Embed}}\ E) \\
&\qquad\qquad\qquad\qquad \mathit{LetList}) \\
\mathbf{data}\ E &= \ldots \\
&\mid\ \mathit{LetStar}\ \mathit{LetList}
\end{aligned}$$

Nested binding?

$$\text{let* } x_1 = e_1, \ldots, x_n = e_n \text{ in } e$$

First try:

$$\begin{array}{lll}
\mathbf{data}\ \mathit{LetList} & = & \mathit{Nil} \\
 & | & \mathit{Binding}\ (\textbf{Bind}\ (N, \textbf{Embed}\ E) \\
 & & \qquad\qquad\qquad \mathit{LetList}) \\
\mathbf{data}\ E & = & \ldots \\
 & | & \mathit{LetStar}\ (\textbf{Bind}\ \mathit{LetList}\ E)
\end{array}$$

Nested binding?

$$\text{let* } x_1 = e_1, \ldots, x_n = e_n \text{ in } e$$

Better:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{LetList} & = & \mathit{Nil} \\
 & \mid & \mathit{Binding}\ (\textsf{\textbf{Rebind}}\ (N, \textsf{\textbf{Embed}}\ E) \\
 & & \qquad\qquad\qquad\qquad \mathit{LetList}) \\
\mathbf{data}\ E & = & \ldots \\
 & \mid & \mathit{LetStar}\ (\textsf{\textbf{Bind}}\ \mathit{LetList}\ E)
\end{array}$$

Want to know more? Read our paper!

On Hackage:
`http://hackage.haskell.org/package/unbound/`

```
cabal install unbound
```