Lecture 7

Substitution and validity

- $E[E'/V]$ **means:**
  - **the result of substituting** E'
  - **for each** *free* **occurrence of** V **in** E**.**
- **The substitution is valid if**
  - **no free variable in** E'
  - **became bound in** $E[E'/V]$
- **In the definitions of** α**- and** β**-conversion, it was stipulated that the substitutions involved must be valid**
  - **for example** $(\lambda V.\ E_1)\ E_2 \underset{\beta}{\longrightarrow} E_1[E_2/V]$
  - **as long as the substitution** $E_1[E_2/V]$ **valid**
- **Convenient to extend the meaning of** $E[E'/V]$ **so that we don't have to worry about validity**
  - **i.e. arrange that** *all* **expressions** E, E_1 **and** E_2 **and** *all* **variables** V **and** V'**:**
    $(\lambda V.\ E_1)\ E_2 \longrightarrow E_1[E_2/V]$ **and** $\lambda V.\ E \longrightarrow \lambda V'.\ E[V'/V]$

Definition of substitution

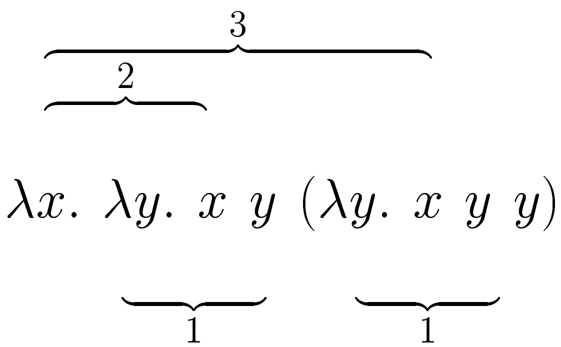
- $E\,[E'/V]$ **defined recursively on structure of** E**:**

E	$E\,[E'/V]$
V	E'
V' **(where** $V \neq V'$**)**	V'
$E_1\ E_2$	$E_1\,[E'/V]\ E_2\,[E'/V]$
$\lambda V.\ E_1$	$\lambda V.\ E_1$
$\lambda V'.\ E_1$ **(where** $V \neq V'$ **and** V' **is not free in** E'**)**	$\lambda V'.\ E_1\,[E'/V]$
$\lambda V'.\ E_1$ **(where** $V \neq V'$ **and** V' **is free in** E'**)**	$\lambda V''.\ E_1\,[V''/V']\,[E'/V]$ **where** V'' **is a variable not free in** E' **or** E_1

$$\begin{aligned}(\lambda y.\ y\ x)\,[y/x] &\equiv \lambda z.\ (y\ x)\,[z/y]\,[y/x]\\ &\equiv \lambda z.\ (z\ x)\,[y/x]\\ &\equiv \lambda z.\ z\ y\end{aligned}$$

De Bruijn terms

- **De Bruijn's idea:**
  - **variables are 'pointers' to the λs that bind them**
- **Can point to the appropriate λ by giving the number of levels 'upwards' needed to reach it**
- $\lambda x.\ \lambda y.\ x\ y$ **is represented by** $\lambda\lambda 2\ 1$
- **Diagram shows number of levels separating a variable from the λ that binds it**



  - **represented by** $\lambda\lambda 2\ 1\ \lambda 3\ 1\ 1$

Representation of free variables

- **Free variables represented by numbers bigger than the depth of λs above them**
  - **different free variables assigned different numbers**
- $\lambda x.\ (\lambda y.\ y\ x\ z)\ x\ y\ w$ **represented by** $\lambda(\lambda 1\ 2\ 3)\ 1\ 2\ 4$
  - **only two λs above the occurrence of** 3
  - **this number must denote a free variable**
  - **similarly there is only one λ above the second occurrence of 2 and the occurrence of 4**
  - **so these too must be free variables**
  - 2 **could not be used to represent** w
    - **since this had already been used to represent the free** y
  - **chose the first available number bigger than** 2
    - 3 **was already in use representing** z

More on free variables

- **Must assign big enough numbers to free variables**
  - **the first occurrence of** z **in** $\lambda x.\ z\ (\lambda y.\ z)$ **could be represented by** 2
  - **but the second occurrence requires** 3
    - **since they are the same variable must use** 3
- **Hence** $\lambda x.\ z\ (\lambda y.\ z)$ **represented by** $\lambda 3 \lambda 3$
- $\lambda x.\ x\ (\lambda y.\ x\ y\ y)$ **represented by** $\lambda 1 (\lambda 2\ 1\ 1)$

The λ-calculus in ML

- **Datatype `lam` to represent λ-expressions**

```
datatype lam = Var of string
             | App of (lam * lam)
             | Abs of (string * lam);
```

- $(\lambda x\ y.\ f\ x\ y)\ z$ **represented by:**

```
App
 (Abs ("x",
       Abs ("y",
             App (App (Var "f", Var "x"), Var "y"))),
  Var "z")
```

Computing free variables

- **Some set-theoretic functions:**

```
fun Member x [] = false
 |  Member x (x'::s) = (x=x') orelse Member x s;

fun Union [] l = l
 |  Union (x::l1) l2 =
     if Member x l2 then Union l1 l2
                    else x::(Union l1 l2);

fun Subtract [] l = []
 |  Subtract (x::l1) l2 =
     if Member x l2 then Subtract l1 l2
                    else x::(Subtract l1 l2);
```

- **Computing the *set* of free variables**

```
fun Frees (Var x) = [x]
 |  Frees (App(e1,e2)) = Union (Frees e1) (Frees e2)
 |  Frees (Abs(x,e)) = Subtract (Frees e) [x];
>  val Frees = fn : lam -> string list

Frees(Abs ("x",App (App (Var "f",Var "x"),Var "y")));
> val it = ["f","y"] : string list
```

Functions for renaming variables:

- **Adding a prime to a variable name**

```
fun Prime x = x^"'";
> val Prime = fn : string -> string

Prime "foo";
> val it = "foo'" : string
```

- **Priming a variable until it is distinct from all variables in a given list**

```
fun Variant xl x =
 if Member x xl then Variant xl (Prime x) else x;
> val Variant = fn : string list -> string -> string

Variant [] "foo";
> val it = "foo" : string

Variant ["bas","foo","mumble"] "foo";
> val it = "foo'" : string

Variant ["bas","foo","mumble","foo'"] "foo";
> val it = "foo''" : string
```

Substitution in ML

E	$E[E'/V]$
V	E'
V' (**where** $V \neq V'$)	V'
$E_1\ E_2$	$E_1[E'/V]\ E_2[E'/V]$
$\lambda V.\ E_1$	$\lambda V.\ E_1$
$\lambda V'.\ E_1$ (**where** $V \neq V'$ **and** V' **is not free in** E')	$\lambda V'.\ E_1[E'/V]$
$\lambda V'.\ E_1$ (**where** $V \neq V'$ **and** V' **is free in** E')	$\lambda V''.\ E_1[V''/V'][E'/V]$ **where** V'' **is a variable not free in** E' **or** E_1

```
fun Subst (e as Var v') e' v = if v=v' then e' else e
 |  Subst (App(e1, e2)) e' v =
      App(Subst e1 e' v, Subst e2 e' v)
 |  Subst (e as Abs(v',e1)) e' v =
     if v=v'
      then e
      else
       if Member v' (Frees e')
        then
         let val v'' = Variant (Frees e' @ Frees e1) v'
         in Abs(v'', Subst(Subst e1 (Var v'') v') e' v)
         end
        else Abs(v', Subst e1 e' v);
```

Representing Things in the λ-calculus

- **λ-calculus appears to be very primitive**
  - **however, it can represent most of the objects and structures needed for programming**

- **Goal: represent objects and structures so they have required properties**

- **For example, to represent**
  - **constants** *true* **and** *false*
  - **Boolean function $\neg$ ('not')**

- **define λ-expressions**
  - `true`, `false` **and** `not`

- **So that:**

$$\texttt{not true} = \texttt{false}$$
$$\texttt{not false} = \texttt{true}$$

Repesenting $\wedge$ ('and') & $\vee$ ('or')

- **To represent Boolean function $\wedge$ ('and')**
- **Define λ-expression `and` such that:**

$$\begin{aligned}\texttt{and true true} &= \texttt{true}\\ \texttt{and true false} &= \texttt{false}\\ \texttt{and false true} &= \texttt{false}\\ \texttt{and false false} &= \texttt{false}\end{aligned}$$

- **To represent $\vee$ ('or')**
- **Define `or` such that:**

$$\begin{aligned}\texttt{or true true} &= \texttt{true}\\ \texttt{or true false} &= \texttt{true}\\ \texttt{or false true} &= \texttt{true}\\ \texttt{or false false} &= \texttt{false}\end{aligned}$$

Notation for definitions

- **λ-expressions used to represent things may appear completely unmotivated**
  - **they are chosen so that they work**

- **Notation: write**

$$\texttt{LET} \sim = \lambda\text{-}\mathit{expression}$$

  to introduce $\sim$ as a new notation

- **Usually $\sim$ is a name like `true` or `and`**
  - **such names are written in `this font` or underlined**
  - true **is a variable, but `true` is $\lambda x.\ \lambda y.\ x$**
  - 2 **is a number, but $\underline{2}$ is $\lambda f\ x.\ f(f\ x)$**
  - **explanation coming ... !**

- **Sometimes $\sim$ will be more complicated**
  - **like the conditional notation $(E \rightarrow E_1 \mid E_2)$**

Representing truth-values (Booleans)

- **Define `true`, `false` and `not` so that:**

$$\texttt{not true} = \texttt{false}$$
$$\texttt{not false} = \texttt{true}$$

$$(\texttt{true} \to E_1 \mid E_2) = E_1$$
$$(\texttt{false} \to E_1 \mid E_2) = E_2$$

  - LET true $= \lambda x.\ \lambda y.\ x$
  - LET false $= \lambda x.\ \lambda y.\ y$
  - LET not $= \lambda t.\ t$ false true

- **Rules of λ-conversion verify this works:**

$$\begin{array}{lll} \texttt{not true} & = (\lambda t.\ t\ \texttt{false true})\ \texttt{true} & \textbf{(defn of } \texttt{not}\textbf{)} \\ & = \texttt{true false true} & \textbf{(}\beta\textbf{-conversion)} \\ & = (\lambda x.\ \lambda y.\ x)\ \texttt{false true} & \textbf{(defn of } \texttt{true}\textbf{)} \\ & = (\lambda y.\ \texttt{false})\ \texttt{true} & \textbf{(}\beta\textbf{-conversion)} \\ & = \texttt{false} & \textbf{(}\beta\textbf{-conversion)} \end{array}$$

- **Similarly** not false = true

Representing conditionals

- **Conditionals** $(E \rightarrow E_1 \mid E_2)$ **defined by**
  - `LET` $(E \rightarrow E_1 \mid E_2) = (E\ E_1\ E_2)$

- **For any** λ**-expressions** E, E_1 **and** E_2
  - $(E \rightarrow E_1 \mid E_2)$ **stands for** $(E\ E_1\ E_2)$

- **The conditional notation behaves as it should:**

$$\begin{aligned}(\texttt{true} \rightarrow E_1 \mid E_2) &= \texttt{true}\ E_1\ E_2 \\ &= (\lambda x\ y.\ x)\ E_1\ E_2 \\ &= E_1\end{aligned}$$

and

$$\begin{aligned}(\texttt{false} \rightarrow E_1 \mid E_2) &= \texttt{false}\ E_1\ E_2 \\ &= (\lambda x\ y.\ y)\ E_1\ E_2 \\ &= E_2\end{aligned}$$