Lecture 9

λ-reduction as evaluation

- **If $E_1 \longrightarrow E_2$**
  - E_2 **got from** E_1 **by 'evaluation'**
  - **If no (β- or η-) redexes in E_2 then it's 'fully evaluated'**
- **a λ-expression is said to be *in normal form* if it contains no β- or η-redexes**
  - **i.e. if the only conversion rule that can be applied is α-conversion**
  - **a λ-expression in normal form is 'fully evaluated'**
- **Examples:**
  - **Church numerals are all in normal form**
  - $(\lambda x.\ x)\ \underline{0}$ **is not in normal form**
- **Can also define 'δ-normal form'**

Church Rosser Theorem

- **Statement of the Church-Rosser theorem:**

  If $E_1 = E_2$ then there exists an E such that $E_1 \longrightarrow E$ and $E_2 \longrightarrow E$

- **Suppose normal forms E_1 and E_2 are obtained from E by sequences of conversions**

  - **hence $E = E_1$ and $E = E_2$**
  - **hence $E_1 = E_2$**
  - **By Church-Rosser theorem there exists an expression E'**
    - $E_1 \longrightarrow E'$ **and** $E_2 \longrightarrow E'$
  - **the only redexes E_1 and E_2 can contain are α-redexes**
  - **so only way that E_1 and E_2 can be reduced to E' is by α-conversion**
  - **so E_1 and E_2 must be the same up to renaming of bound variables**

Parallel evaluation

- **Suppose E is 'evaluated' in two different ways by applying different sequences of reductions until normal forms E_1 and E_2 are obtained**

- **The *Church-Rosser theorem* shows that E_1 and E_2 will be the same**

  - **up to α-conversion**

  - **i.e. except for having possibly different names of bound variables**

- **Because the results of reductions do not depend on the order in which they are done, separate redexes can be evaluated in parallel**

  - **suggests multiprocessor achitectures**

  - **distributing redexes to processors and collecting results may cancel out theoretical advantages**

Church numerals are not equal

- **Suppose** $m \neq n$ **but** $\underline{m} = \underline{n}$

- **By the Church-Rosser theorem** $\underline{m} \longrightarrow E$ **and** $\underline{n} \longrightarrow E$ **for some** E

- **Consider definitions of** $\underline{m}$ **and** $\underline{n}$

$$\underline{m} = \lambda f\ x.\ f^m\ x$$
$$\underline{n} = \lambda f\ x.\ f^n\ x$$

- **no such** E **can exist**

- **only conversions applicable to** $\underline{\mathbf{m}}$ **and** $\underline{\mathbf{n}}$ **are** α**-conversions**

- **these cannot change the number of function applications in an expression**
  ($\underline{m}$ **contains** m **applications and** $\underline{n}$ **contains** n **applications)**

Corollaries to Church-Rosser Theorem

- **Definition:** E *has a normal form* **if** $E = E'$ **for some** E' **in normal form**
- **If** E **has a normal form then** $E \longrightarrow E'$ **for some** E' **in normal form**
  - **If** E **has a normal form then** $E = E'$ **for some** E' **in normal form**
  - **by Church-Rosser theorem there exists** E'' **such that** $E \longrightarrow E''$ **and** $E' \longrightarrow E''$
  - **as** E' **in normal form only redexes in it are** α**-redexes**
  - **so reduction** $E' \longrightarrow E''$ **must consist only of of** α**-conversions**
  - **thus** E'' **must be identical to** E' **except for renaming of bound variables**
    - **it must thus be in normal form as** E' **is**

Corollaries to CR continued

- **If E has a normal form and $E = E'$ then E' has a normal form**
  - suppose E has a normal form and $E = E'$
  - As E has a normal form, $E = E''$ where E'' is in normal form
  - hence $E' = E''$ by the transitivity of $=$
  - so E' has a normal form
- **If $E = E'$ and E and E' are both in normal form, then E and E' are identical up to α-conversion**
  - by Church-Rosser there exists E'' such that $E \longrightarrow E''$ and $E' \longrightarrow E''$
  - if E and E' are in normal form, then reductions to E'' must be α-reductions
  - so E and E' are convertable to each other via α-conversions

Exercises

- **For each of the following *either* find its normal form *or* show that it has no normal form:**

  (i) $\texttt{add}\ \underline{3}$

  (ii) $\texttt{add}\ \underline{3}\ \underline{5}$

  (iii) $(\lambda x.\ x\ x)\ (\lambda x.\ x)$

  (iv) $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

  (v) $\texttt{Y}$

  (vi) $\texttt{Y}\ (\lambda y.\ y)$

  (vii) $\texttt{Y}\ (\lambda f\ x.\ (\texttt{iszero}\ x \to \underline{0} \mid\ f\ (\texttt{pre}\ x)))\ \underline{7}$

Non-termination

- **A λ-expression E can have a normal form**
  - **even if there's an infinite sequence** $E \longrightarrow E_1 \longrightarrow E_2 \ \cdots$

- **Example:**
  - $(\lambda x.\ \underline{1})\ (\mathtt{Y}\ f)$ **has a normal form** $\underline{1}$
  - **even though:**

$$(\lambda x.\ \underline{1})\ (\mathtt{Y}\ f) \longrightarrow (\lambda x.\ \underline{1})\ (f\ (\mathtt{Y}\ f)) \longrightarrow \cdots (\lambda x.\ \underline{1})\ (f^n\ (\mathtt{Y}\ f)) \longrightarrow \cdots$$

Normalisation theorem

- **If E has a normal form, then**
  - **repeatedly reducing the leftmost β- or η-redex will terminate with an expression in normal form**

- **Normalisation theorem gives an algorithm for computing normal forms (when they exist)**

- **A sequence of reductions in which the leftmost redex is always reduced is called a** *normal order reduction sequence*

- **Normalization theorem says that**
  - **if E has a normal form**
  - **then it is got by normal order reduction**

Inefficiencies

- **Normal order reduction often inefficient**
- **Example: by normal order reduction:**

$$(\lambda x. \smile x \smile x \smile)\ E$$

is reduced to

$$\smile E \smile E \smile$$

  - **suppose E is not in normal form**
  - **more efficient to first reduce E to normal form E'**
  - **then reduce**

$$(\lambda x. \smile x \smile x \smile)\ E'$$

to

$$\smile E' \smile E' \smile$$

  - **avoid reducing E twice**
  - **this is what ML does**

Call-by-Value

- **ML reduces arguments before substituting**
  - **disastrous in cases like** $(\lambda x.\underline{1})\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$
- **Difficult problem to a find an optimal algorithm for choosing the next redex to reduce**
- **Call-by-value is appropriate when the language has constructs with side effects**
  - **e.g. assignments, as in ML**
- **Normal order evaluation is not as inefficient as one might think**
  - **cunning implementation tricks like graph reduction**
- **Whether functional programming languages should use normal order or call by value is still a controversial issue**

On 'undefined' λ-expressions

- E_1 **may not have a normal form even though** $E_1\ E_2$ **does have one**

- **Example**

  - $\mathtt{Y}$ **has no normal form,**

  - **but** $\mathtt{Y}\ (\lambda x.\ \underline{1}) \longrightarrow \underline{1}$

- λ**-expressions without a normal form are not 'undefined' functions**

  - $\mathtt{Y}$ **has no normal form but it denotes a perfectly well defined function**

Head normal form

- **A λ-expression denotes an undefined function if and only if it *cannot* be converted to an expression in *head normal form***

- **E is in head normal form if it has the form**

$$\lambda V_1 \ \cdots \ V_m.\ V \ E_1 \ \cdots \ E_n$$

  - **where $V_1, \ldots, V_m$ and V are variables**
  - **and $E_1, \ldots, E_n$ are λ-expressions**
  - **V can either be equal to V_i, for some i, or it can be distinct from all of them**

Definedness of `Y`

- `Y` **is not undefined because it can be converted to**

$$\lambda f.\ f\ ((\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)))$$

  - **this is in head normal form**

- **Can be shown that an expression E has a head normal form**
  - **if and only if there exist expressions $E_1, \ldots, E_n$**
  - **such that $E\ E_1\ \ldots\ E_n$ has a normal form**

- **This supports the interpretation of expressions without head normal forms as denoting undefined functions**
  - E **being undefined means that $E\ E_1\ \ldots\ E_n$ never terminates for *any* $E_1, \ldots, E_n$**

Programming reduction in ML

- **Recall**

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

- $E[E'/V]$ **computed by** `Subst E E' V`

- **Normal order reduction in ML**

```
fun EvalN (e as Var _ ) = e
 |  EvalN (Abs(x,e))    = Abs(x, EvalN e)
 |  EvalN (App(e1,e2))  =
     case EvalN e1
     of   (Abs(x,e3)) => EvalN(Subst e3 e2 x)
     |    e1'         => App(e1', EvalN e2);
> val EvalN = fn : lam -> lam
```

Applicative (call-by-value) order

- **With call-by-value, function bodies are not evaluated**

```
fun EvalV (e as Var _)    = e
 |  EvalV (e as Abs(_,_)) = e
 |  EvalV (App(e1,e2))    =
     let val e2' = EvalV e2
     in
     (case EvalV e1
      of   (Abs(x,e3)) => EvalV(Subst e3 e2' x)
      |    e1'         => App(e1',e2'))
     end;
>  EvalV = fn : lam -> lam
```