Lecture 10
Reduction with $\delta$-rules

- Assume as primitive constants (atoms):
  - integers
  - unary operators
  - binary operators

- atom packages these into a single datatype

- unary operators and binary operators have:
  - a name
  - a semantics – ML function coding a $\delta$ rule

```
datatype atom = Num of int
  | Op1 of string * (int->int)
  | Op2 of string * (int*int->int);
```
• Application of atomic operation to a value defined by ConApply
  • computes $\delta$-reduction

• Application of a binary operator $b$ to $m$
  • results in a unary operator named $m b$
  • expecting the other argument

• So for each binary operator $b$ and number $m$
  there will be a unary operator named $m b$
  • allows all $\delta$-rules to be binary:
    \[
    b \ m \rightarrow^\delta b m
    \]
  • need to compute name of $b m$ by concatenating name of $b$ with name of $m$
Converting numbers to strings

- Need to convert number $m$ to a string
  - for concatenation with the name of operator

```haskell
fun StringOfNum 0 = "0"
| StringOfNum 1 = "1"
| StringOfNum 2 = "2"
| StringOfNum 3 = "3"
| StringOfNum 4 = "4"
| StringOfNum 5 = "5"
| StringOfNum 6 = "6"
| StringOfNum 7 = "7"
| StringOfNum 8 = "8"
| StringOfNum 9 = "9"
| StringOfNum n =
  (StringOfNum(n div 10)) ^ (StringOfNum(n mod 10));

StringOfNum 1574;
> val it = "1574" : string
```
Definition of conapply

fun ConApply(Op1(_,f1), Num m) = Num(f1 m)
  | ConApply(Op2(x,f2), Num m) = 
      Op1((StringOfNum m^x), fn n => f2(m,n));
> val ConApply = fn : atom * atom -> atom

ConApply(Op2("+",op +), Num 2);
> val it = Op1 ("2+",fn) : atom

ConApply(it, Num 3);
> val it = Num 5 : atom
λ-calculus with constants (atoms)

- Redefine lam

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

- Normal order evaluation with δ-rules

```haskell
fun EvalN (e as Var _) = e
  | EvalN (e as Con _) = 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' as Con a1)
        => (case EvalN e2
            of (Con a2) => Con(ConApply(a1,a2))
              | e2' => App(e1',e2'))
        | e1'
        => App(e1', EvalN e2);
> val EvalN = fn : lam -> lam
```

- Consider \( \text{App(Num 1, Num2)} \) ...
Call-by-value with δ-rules

fun EvalV (e as Var _) = e
| EvalV (e as Con _) = 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' as Con a)
      => (case e2'
              of (Con a2) => Con(ConApply(a1,a2))
                  | _ =⇒ App(e1',e2'))
    | e1'
      => App(e1',e2'))
  end;
Representing the recursive functions

- **Recursive functions** are an important class of numerical functions

- Shortly after Church invented the $\lambda$-calculus, Kleene proved that every recursive function could be represented in it

- This provided evidence for *Church’s thesis*
  
  - the hypothesis that any intuitively computable function could be represented in the $\lambda$-calculus
  
  - has been shown that many other models of computation define the same class of functions that can be defined in the $\lambda$-calculus.
  
  - e.g. Turing machines
Representing a numerical function

- Number \( n \) is represented by the \( \lambda \)-expression \( n \)

- \( \lambda \)-expression \( f \) represents function \( f \) iff
  - for all numbers \( x_1, \ldots, x_n \):
    \[
    f(x_1, \ldots, x_n) = y \quad \text{if} \quad f(x_1, \ldots, x_n) = y
    \]

- A function is \textit{primitive recursive} if it can be constructed by a finite sequence of applications of the operations of substitution and primitive recursion starting from 0, \( S \) and the projection functions \( U^n_i \) (all defined below)
Base functions and Substitution

- **Successor function** $S$:
  - $S(x) = x + 1$

- **Projection functions** $U^i_n$ ($n$ and $i$ are numbers):
  - $U^i_n(x_1, x_2, \ldots, x_n) = x_i$

- **Suppose**:
  - $g$ is a function of $r$ arguments
  - $h_1, \ldots, h_r$ are $r$ functions each of $n$ arguments

- **We say** $f$ is defined from $g$ and $h_1, \ldots, h_r$ by substitution if:
  $$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$
Primitive recursion

- Suppose:
  - $g$ is a function of $n-1$ arguments
  - $h$ is a function of $n+1$ arguments

- Then $f$ is defined from $g$ and $h$ by primitive recursion if:
  
  \[
  f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \\
  f(S(x_1), x_2, \ldots, x_n) = h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n)
  \]

  - $g$ is called the base function
  - $h$ is called the step function

- Primitive Recursion Theorem:
  
  - Can proved that for any base and step function there always exists a unique function defined from them by primitive recursion

- Addition function $\text{sum}$ is primitive recursive:
  
  \[
  \text{sum}(0, x_2) = x_2 \\
  \text{sum}(S(x_1), x_2) = S(\text{sum}(x_1, x_2))
  \]
PR functions in $\lambda$-calculus

- Obvious that:
  - $0$ represents $0$
  - $\text{suc}$ represents $S$
  - $\lambda p. p \downarrow i$ represents $U^n_i$

- Suppose
  - function $g$ of $r$ variables is represented by $g$
  - functions $h_i$ ($1 \leq i \leq r$) of $n$ variables represented by $h_i$

- Then if a function $f$ of $n$ variables is defined by substitution by:
  
  $$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$

  then $f$ is represented by $f$ where:

  $$f = \lambda(x_1, \ldots, x_n). g(h_1(x_1, \ldots, x_n), \ldots, h_r(x_1, \ldots, x_n))$$
Suppose $f$ of $n$ variables is defined inductively

- from a base function $g$ of $n-1$ variables
  and an inductive step function $h$ of $n+1$ variables

- then

$$f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n)$$
$$f(S(x_1), x_2, \ldots, x_n) = h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n)$$

Thus if $g$ represents $g$ and $h$ represents $h$ then $f$ will represent $f$ if

$$f(x_1, x_2, \ldots, x_n) =$$
$$\begin{cases}
\text{iszero } x_1 \\
\rightarrow g(x_2, \ldots, x_n) \\
| h(f(\text{pre } x_1, x_2, \ldots, x_n), \text{pre } x_1, x_2, \ldots, x_n))
\end{cases}$$

A solution to this equation is:

$$Y(\lambda f. \lambda(x_1, x_2, \ldots, x_n).$$
$$\begin{cases}
\text{iszero } x_1 \\
\rightarrow g(x_2, \ldots, x_n) \\
| h(f(\text{pre } x_1, x_2, \ldots, x_n), \text{pre } x_1, x_2, \ldots, x_n))
\end{cases}$$

Primitive recursive functions are representable
The recursive functions

• A function is called *recursive*
  - if it can be constructed from 0, the successor function and the projection functions
  - by a sequence of substitutions, primitive recursions
  - and *minimizations*

• Suppose $g$ is a function of $n$ arguments
  - $f$ is defined from $g$ by minimization if:
    \[ f(x_1, x_2, \ldots, x_n) = \text{‘the smallest } y \text{ such that } g(y, x_2, \ldots, x_n) = x_1 \text{’} \]

• MIN($f$) denotes the minimization of $f$
Undefinedness

- Functions defined by minimization may be undefined for some arguments

- For example, if \( \text{one} \) is the function that always returns 1
  - i.e. \( \text{one}(x) = 1 \) for every \( x \)

- \( \text{MIN}(\text{one}) \) is only defined for arguments with value 1

- Obvious because if \( f(x) = \text{MIN}(\text{one})(x) \), then:
  \[
  f(x) = \text{‘the smallest } y \text{ such that } \text{one}(y)=x\text{’}
  \]
  and clearly this is only defined if \( x = 1 \)

- Thus
  \[
  \text{MIN}(\text{one})(x) = \begin{cases} 
  0 & \text{if } x = 1 \\
  \text{undefined} & \text{otherwise}
  \end{cases}
  \]
Representing minimisation

- Suppose \( g \) represents a function \( g \) of \( n \) variables and \( f \) is defined by \( f = \text{MIN}(g) \).

- If a \( \lambda \)-expression \( \text{min} \) can be devised such that
  \[
  \text{min} \ x \ f \ (x_1, \ldots, x_n)
  \]
  represents least \( y \) greater than \( x \) such that
  \[
  f(y, x_2, \ldots, x_n) = x_1
  \]
  then \( g \) will represent \( g \) where:
  \[
  g = \lambda(x_1, x_2, \ldots, x_n). \text{min} \ 0 \ f \ (x_1, x_2, \ldots, x_n)
  \]

- \( \text{min} \) will have the desired property if:
  \[
  \text{min} \ x \ f \ (x_1, x_2, \ldots, x_n) =
  \begin{cases} 
  (\text{eq} \ (f(x, x_2, \ldots, x_n)) \ x_1) & \rightarrow x \ | \ \text{min} \ (\text{suc} \ x) \ f \ (x_1, x_2, \ldots, x_n) \\
  \end{cases}
  \]
  \[
  (\text{eq} \ m \ n = \text{true} \ \text{if} \ m = n, \ \text{eq} \ m \ n = \text{false} \ \text{if} \ m \neq n)
  \]

- Thus \( \text{min} \) can simply be defined to be:
  \[
  Y(\lambda m. \ \\
  \lambda x \ f \ (x_1, x_2, \ldots, x_n). \ \\
  (\text{eq} \ (f(x, x_2, \ldots, x_n)) \ x_1 \ \\
  \rightarrow x \ | \ m \ (\text{suc} \ x) \ f \ (x_1, x_2, \ldots, x_n))))
  \]
Higher-order primitive recursion

- Ackermann’s function, $\psi$, is recursive but not primitive recursive

  \[
  \begin{align*}
  \psi(0, n) & = n+1 \\
  \psi(m+1, 0) & = \psi(m, 1) \\
  \psi(m+1, n+1) & = \psi(m, \psi(m+1, n))
  \end{align*}
  \]

- If one allows functions as arguments, then many more recursive functions can be defined by a primitive recursion

- Define $\text{rec}$ by primitive recursion as follows:

  \[
  \begin{align*}
  \text{rec}(0, x_2, x_3) & = x_2 \\
  \text{rec}(S(x_1), x_2, x_3) & = x_3(\text{rec}(x_1, x_2, x_3))
  \end{align*}
  \]

- Then $\psi$ can be defined by:

  \[
  \psi(m, n) = \text{rec} (m, S, f \mapsto (x \mapsto \text{rec}(x, f(1), f))) (n)
  \]

  - where $x \mapsto \theta(x)$ maps $x$ to $\theta(x)$
  - the third argument of $\text{rec}$, $x_3$, is a function
  - in the definition of $\psi$, $x_2$ is a function, viz. $S$
Power of higher-order recursion

- A function which takes another function as an argument, or returns another function as a result, is called \textit{higher-order}.

- The example $\psi$ shows that higher-order primitive recursion is more powerful than ordinary primitive recursion.

- Operators like \texttt{rec} make functional programming very powerful.
A partial function is one that is not defined for all arguments

- the function MIN\((one)\) described above is partial
- the division function is also partial, since division by 0 is not defined

Functions that are defined for all arguments are called total

A partial function is partial recursive if it can be constructed from 0, the successor function and the projection functions by a sequence of substitutions, primitive recursions and minimizations

- thus the recursive functions are just the partial recursive functions which happen to be total

Can be shown that every partial recursive function \(f\) can be represented by a \(\lambda\)-expression \(\bar{f}\) in the sense that

(i) \(\bar{f}(x_1, \ldots, x_n) = y\) if \(f(x_1, \ldots, x_n) = y\)

(ii) If \(f(x_1, \ldots, x_n)\) is undefined then \(\bar{f}(x_1, \ldots, x_n)\) has no normal form.