Lecture 10

Reduction with δ-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 δ rule**

```
datatype atom = Num of int
              | Op1 of string * (int->int)
              | Op2 of string * (int*int->int);
```

`conapply`

- **Application of atomic operation to a value defined by `ConApply`**
  - **computes δ-reduction**

- **Application of a binary operator `b` to `m`**
  - **results in a unary operator named `mb`**
  - **expecting the other argument**

- **So for each binary operator `b` and number `m` there will be a unary operator named `mb`**
  - **allows all δ-rules to be binary:**

$$\mathtt{b\ m} \xrightarrow[\delta]{} \mathtt{bm}$$

  - **need to compute name of `bm` 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**

```
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`

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

- **Normal order evaluation with δ-rules**

```
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** `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 λ-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 λ-calculus

  - has been shown that many other models of compution define the same class of functions that can be defined in the λ-calculus.

  - e.g. Turing machines

Representing a numerical function

- **Number** n **is represented by the** λ**-expression** $\underline{n}$

- λ**-expression** $\underline{f}$ **represents function** f **iff**

  - **for all numbers** x_1**,** $\ldots$**,** x_n**:**

$$\underline{f}(\underline{x_1}, \ldots, \underline{x_n}) = \underline{y} \quad \textbf{if} \quad f(x_1, \ldots, x_n) = y$$

- **A function is** *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_n^i (n and i are numbers):**
  - $U_n^i(x_1, x_2, \ldots, x_n) = x_i$
- **Suppose:**
  - g **is a function of** r **arguments**
  - h_1, ... , h_r **are** r **functions each of** n **arguments**
- **We say f is defined from g and h_1, ... , 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:**

$$\begin{aligned} f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(S(x_1), x_2, \ldots, x_n) &= h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n) \end{aligned}$$

  - 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** *sum* **is primitive recursive:**

$$\begin{aligned} sum(0, x_2) &= x_2 \\ sum(S(x_1), x_2) &= S(sum(x_1, x_2)) \end{aligned}$$

PR functions in λ-calculus

- **Obvious that:**
  - $\underline{\mathbf{0}}$ **represents** 0
  - $\mathtt{suc}$ **represents** S
  - $\lambda p.\ p \overset{n}{\downarrow} i$ **represents** U_n^i
- **Suppose**
  - **function** g **of** r **variables is represented by** $\mathtt{g}$
  - **functions** h_i $(1 \leq i \leq r)$ **of** n **variables represented by** $\mathtt{h}_i$
- **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** $\mathtt{f}$ **where:**

  $$\mathtt{f} = \lambda(x_1, \ldots, x_n).\ \mathtt{g}(\mathtt{h}_1(x_1, \ldots, x_n), \ldots, \mathtt{h}_r(x_1, \ldots, x_n))$$

Representing Primitive Recursion

- **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**

$$
\begin{aligned}
f(0, x_2, \ldots, x_n) &= g(x_2, \ldots, x_n) \\
f(S(x_1), x_2, \ldots, x_n) &= h(f(x_1, x_2, \ldots, x_n), x_1, x_2, \ldots, x_n)
\end{aligned}
$$

- **Thus if $\texttt{g}$ represents g and $\texttt{h}$ represents h then $\texttt{f}$ will represent f if**

$$
\begin{array}{l}
\texttt{f}\ (x_1, x_2, \ldots, x_n) = \\
\quad (\texttt{iszero}\ x_1 \\
\qquad \rightarrow \texttt{g}(x_2, \ldots, x_n) \\
\qquad \mid\ \texttt{h}(\texttt{f}\ (\texttt{pre}\ \ x_1, x_2, \ldots, x_n), \texttt{pre}\ \ x_1, x_2, \ldots, x_n))
\end{array}
$$

- **A solution to this equation is:**

$$
\begin{array}{l}
\texttt{Y}(\lambda f.\ \lambda(x_1, x_2, \ldots, x_n). \\
\qquad\quad (\texttt{iszero}\ x_1 \\
\qquad\qquad \rightarrow \texttt{g}(x_2, \ldots, x_n) \\
\qquad\qquad \mid\ \texttt{h}(f(\texttt{pre}\ \ x_1, x_2, \ldots, x_n), \texttt{pre}\ \ x_1, x_2, \ldots, x_n)))
\end{array}
$$

- **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** *one* **is the function that always returns** 1
  - **i.e.** $one(x) = 1$ **for every** x

- **MIN(***one***) is only defined for arguments with value** 1

- **Obvious because if** $f(x) = \mathrm{MIN}(one)(x)$**, then:**

$$f(x) \quad = \quad \text{`the smallest } y \text{ such that } one(y){=}x\text{'}$$

  and clearly this is only defined if $x = 1$

- **Thus**

$$\mathrm{MIN}(one)(x) = \begin{cases} \underline{0} & \textbf{if } x = 1 \\ \text{undefined} & \textbf{otherwise} \end{cases}$$

Representing minimisation

- **Suppose** `g` **represents a function** g **of** n **variables and** f **is defined by** $f = \mathrm{MIN}(g)$

- **If a** λ**-expression** `min` **can be devised such that**

$$\texttt{min}\ \underline{x}\ \texttt{f}\ (\underline{x}_1, \ldots, \underline{x}_n)$$

represents least y **greater than** x **such that**

$$f(y, x_2, \ldots, x_n) = x_1$$

then `g` **will represent** g **where:**

$$\texttt{g} \ = \ \lambda(x_1, x_2, \ldots, x_n).\ \texttt{min}\ \underline{0}\ \texttt{f}\ (x_1, x_2, \ldots, x_n)$$

- `min` **will have the desired property if:**

$$\begin{array}{l} \texttt{min}\ x\ f\ (x_1, x_2, \ldots, x_n) = \\ \quad (\texttt{eq}\ (f(x, x_2, \ldots, x_n))\ x_1) \\ \qquad \to x \ \mid\ \texttt{min}\ (\texttt{suc}\ x)\ f\ (x_1, x_2, \ldots, x_n)) \end{array}$$

(`eq` $\underline{m}$ $\underline{n}$ $=$ `true` **if** $m{=}n$**,** `eq` $\underline{m}$ $\underline{n}$ $=$ `false` **if** $m{\neq}n$**)**

- **Thus** `min` **can simply be defined to be:**

$$\begin{array}{l} \texttt{Y}(\lambda m. \\ \quad \lambda x\ f\ (x_1, x_2, \ldots, x_n). \\ \qquad (\texttt{eq}\ (f(x, x_2, \ldots, x_n))\ x_1 \\ \qquad\quad \to x \ \mid\ m\ (\texttt{suc}\ x)\ f\ (x_1, x_2, \ldots, x_n))) \end{array}$$

Higher-order primitive recursion

- **Ackermann's function, ψ, is recursive but not primitive recursive**

$$\begin{aligned}\psi(0,n) &= n{+}1\\ \psi(m{+}1,0) &= \psi(m,1)\\ \psi(m{+}1,n{+}1) &= \psi(m,\psi(m{+}1,n))\end{aligned}$$

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

- **Define *rec* by primitive recursion as follows:**

$$\begin{aligned}\mathit{rec}(0,x_2,x_3) &= x_2\\ \mathit{rec}(S(x_1),x_2,x_3) &= x_3(\mathit{rec}(x_1,x_2,x_3))\end{aligned}$$

- **Then ψ can be defined by:**

$$\psi(m,n) \;=\; \mathit{rec}\ (m,\ S,\ f\mapsto(x\mapsto \mathit{rec}(x,f(1),f)))\ (n)$$

  - **where $x\mapsto\theta(x)$ maps x to $\theta(x)$**
  - **the third argument of *rec* , x_3, is a function**
  - **in the definition of ψ, 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** *higher-order*

- **The example ψ shows that higher-order primitive recursion is more powerful than ordinary primitive recursion**

- **Operators like** *rec* **make functional programming very powerful**

The partial recursive functions

- **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 λ-expression $\underline{f}$ in the sense that**

  (i) $\underline{f}(\underline{x_1}, \ldots, \underline{x_n}) = \underline{y}$ **if** $f(x_1, \ldots, x_n) = y$

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