cs611 Lecture 22 **Recursion & Substitution in Lambda Calculus** 21 October, 2002

Scribe: Ashwin Machanavajjhala and Brian Hackett Lecturer: Radu Rugina

1 Lists in Lambda Calculus

The list is inductively defined using the *nil* element and the *cons* operator, which constructs a larger list from a *head* and a *tail*.

$$\begin{array}{rcll} nil & ::= & pair\, true\, true & \text{Here the first field of the pair is true} \\ & & & \text{if the pair is the last element of the list} \\ cons & ::= & \lambda h.\lambda t.\, pair\, false\, (pair\, h\, t) & \\ head & ::= & \lambda c.\, fst\, (snd\, c) & \\ tail & ::= & \lambda c.\, snd\, (snd\, c) & \\ isnill & ::= & \lambda c.\, fst\, c & \end{array}$$

2 Recursion

Can recursive functions be built in λ-Calculus when functions are not named?

$$fact \;=\; \lambda n.test\ (testz\ n)\ c_1\ (mul\ n\ (fact\ (dec\ n))) \tag{1}$$

fact is shorthand notation for a λ-Calculus term, not an actual variable, so this is a recursive equation rather than a definition.

Strategy: Design a higher order function g that makes a successive approximation of a recursive function f, and a third function that computes the fixed point of g.

$$g \;=\; \lambda f.\lambda n.test\ (testz\ n)\ c_1\ (mul\ n\ (f\ \ (dec\ n))) \tag{2}$$

The Y combinator evaluates to the fixed point of its argument.

$$Y \;=\; \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)) \tag{3}$$

$$\begin{array}{rcl} Y\ F & = & (\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ F \\ & \to_1 & (\lambda x.F(x\ x))\ (\lambda x'.F(x'\ x')) \\ & \to_1 & F((\lambda x'.F(x'\ x'))\ (\lambda x'.F(x'\ x'))) \\ & = & F(Y\ F) \end{array}$$

Does Y always terminate if there is a fixed point? Try g = λx.c_1:

$$\begin{array}{rcl} Y\ g & = & (\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))\ g \\ & \to_1 & (\lambda x.g(x\ x))\ (\lambda x'.g(x'\ x')) \\ & \to_1 & g((\lambda x'.g(x'\ x'))\ (\lambda x''.g(x''\ x''))) \\ & \to_1^{CBN} & c_1 \\ or & \to_1^{CBV} & g(g((\lambda x''.g(x''\ x''))\ (\lambda x''.g(x''\ x'')))) \end{array}$$

Y g terminates when CBN is used, but not when CBV is. The problem is in preventing evaluation of (x x): use the equivalent expression λy.x x y instead.

$$Y_{CBV} \;=\; \lambda f.(\lambda x.f(\lambda y.x\ x\ y))\ (\lambda x.f(\lambda y.x\ x\ y)) \tag{4}$$

$$\begin{aligned}
Y_{CBV}\ g &= (\lambda f.(\lambda x.f(\lambda y.x\ x\ y))\ (\lambda x.f(\lambda y.x\ x\ y)))\ g \\
&\rightarrow_1 (\lambda x.g(\lambda y.x\ x\ y))\ (\lambda x'.g(\lambda y'.x'\ x'\ y')) \\
&\rightarrow_1 g(\lambda y.(\lambda x'.g(\lambda y'.x'\ y'\ y'))\ (\lambda x'.g(\lambda y'.x'\ y'\ y'))\ y) \\
&\rightarrow_1 c_1
\end{aligned}$$

$$\begin{aligned}
Y_{CBV}\ F &= (\lambda f.(\lambda x.f(\lambda y.x\ x\ y))\ (\lambda x.f(\lambda y.x\ x\ y)))\ F \\
&\rightarrow_1 (\lambda x.F(\lambda y.x\ x\ y))\ (\lambda x'.F(\lambda y'.x'\ x'\ y')) \\
&\rightarrow_1 F(\lambda y.(\lambda x'.F(\lambda y'.x'\ y'\ y'))\ (\lambda x'.F(\lambda y'.x'\ y'\ y'))\ y) \\
&= F(\lambda y.(Y_{CBV}\ F)\ y) \\
&\approx F(Y_{CBV}\ F)
\end{aligned}$$

Again, using Y_{CBV},

$$\begin{aligned}
g &= \lambda f.\lambda n.test\ (testz\ n)\ c_1\ (mul\ n\ (f\ (dec\ n))) \\
fact &= Y_{CBV}\ g
\end{aligned}$$

Does $(fact\ c_3) \rightarrow c_6$? When using CBV, test evaluates all arguments regardless of the branch taken, so this results in non-termination.

$$g' \;=\; \lambda f.\lambda n.(test\ (testz\ n)\ \lambda x.c_1\ (\lambda x.mul\ n\ (f\ (dec\ n))))\ c_0 \tag{5}$$

Since each branch of test is a function, no β-reductions can be performed on them. After test returns the appropriate branch, the dummy value c_0 is applied to yield the result. $(Y_{CBV} g')$ terminates under both CBN and CBV.

3 Substitution

$$(\lambda x.\ e_0)\ e_1 \longrightarrow_\beta e_0[x \rightarrow e_1]$$

Issues to be considered while substituting:

1. What occurences of x to replace ?
   For instance, consider $x\ (\lambda x.\ x)[x \rightarrow yz]$. Note that this can be rewritten as $x\ (\lambda x'.\ x')[x \rightarrow yz]$ nd hence, only the *free variable* x in e_0 should be substituted.

2. Variable Capture
   A variable which is free before the substitution should not become bound after a substitution. For instance, $(\lambda y.\ x)$ is a constant function. $(\lambda y.\ x)[x \rightarrow y]$ should not result in $(\lambda y.\ y)$ (which is the identity function).

The **Set of Free Variables** in an expression e, denoted by $FV(e)$ is inductively defined as follows:

$$\begin{array}{rcl} FV(x) & ::= & \{x\} \\ FV(e_0\, e_1) & ::= & FV(e_0) \cup FV(e_1) \\ FV(\lambda x.\ e) = FV(e) - \{x\} & & \end{array}$$

In consonance to the specifications and definitions laid down above, substitution can be defined inductively as:

<u>Variables</u>

$$\begin{array}{rcll} x[x \to e] & = & e & \\ y[x \to e] & = & y & \text{if } y \neq x \end{array}$$

<u>Application</u>

$$(e_0\, e_1)[x \to e] = (e_0[x \to e])(e_1[x \to e])$$

<u>Abstraction</u>

$$\begin{array}{rcll} (\lambda x.\ e_0)[x \to e] & = & \lambda x.e_0 & \\ (\lambda y.\ e_0)[x \to e] & = & \lambda y.\ (e_0[x \to e]) & \text{if } (x \neq y \wedge \notin FV(e)) \\ (\lambda y.\ e_0)[x \to e] & = & (\lambda y'.\ e_0[y \to y'])[x \to e] & \text{if } (x \neq y \wedge y \in FV(e) \wedge y' \notin FV(e) \cup FV(e_0)) \end{array}$$