CS611 Lecture 24 **Introduction To FUN** 24 October 2003

Scribe: Adam Arbree Lecturer: Radu Rugina

# 1 Finishing the Lambda Calculus

## 1.1 The Church-Rosser Theorem

$$\forall e_0, e_1, e_2.\ \left[(e_0 \to_\beta^* e_1 \wedge e_0 \to_\beta^* e_2) \Rightarrow \exists e_3.\ (e_1 \to_\beta^* e_3 \wedge e_2 \to_\beta^* e_3)\right]$$

The reductions in the theorem can be viewed as a diagram.

$$\begin{tikzcd}
 & e_0 \arrow[dl] \arrow[dr] & \\
e_1 \arrow[dr] & & e_2 \arrow[dl] \\
 & e_3 &
\end{tikzcd}$$

As the shape of the diagram hints, the Church-Rosser Theorem is also called the the diamond (confluence) property of $\beta$-reduction. A corollary to the Church-Rosser Theorem states that if an expression has a normal form, it is unique. Similar results hold for the $\eta$-reduction and the $\beta\eta$-reduction.

## 1.2 Nameless Representations of the Lambda Calculus

The existance of $\alpha$-renaming implies that the names of variables given in our calculus do not have a significant meaning since they can be changed arbitrarily. We can they consider alternate represenations of the calulus that don't require naming. We consider two alternate methods of represeting the abstraction:

$$\lambda x.\,(\lambda y.\,yx)\,x$$

### 1.2.1 Diagrams

Symbols under and over the ends of the lines are matched in the abstractions.

$$\lambda\Diamond.\,(\lambda\Diamond.\,\Diamond\Diamond)\,\Diamond$$

### 1.2.2 deBuijn Representation

In this representation, variables are replaced with an integer, $k$, such that when $k$ appears it is bound $k$th enclosing abstraction.

$$\lambda.\,(\lambda.\,01)\,0$$

# 2 The FUN language

We define a new functional language that incorporates more complex operations. The FUN language, like REC, also includes recursion, like IMP, has truth values, and, like the lambda calculus, is defined only by expressions and contains higher-order functions.

$$\begin{aligned}
e := \quad & n \mid \mathsf{true} \mid \mathsf{false} \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_1 \mid e_0 \le e_1 \mid e_0 = e_1 \mid \\
& \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \mid \langle e_0, e_1 \rangle \mid \mathsf{fst\ e} \mid \mathsf{snd\ e} \mid \\
& x \mid \lambda x.\,e \mid e_0\ e_1 \mid \mathsf{rec\ y\ (\lambda x.\,e)}
\end{aligned}$$

We can define a Call-By-Value, left-to-right evaluation, small-step operational semantics for FUN[1].

$$\frac{e_1 \to_1 e_1'}{e_0 e_1 \to_1 e_0 e_1'} \qquad \frac{e_0 \to_1 e_0'}{e_0 v \to_1 e_0' v} \qquad \frac{}{(\lambda x.\, e)v \to_1 e[x \mapsto v]} \quad \text{where } v = \lambda x.\, e$$

$$\frac{e_0 \to_1 e_0'}{e_0 \oplus e_1 \to_1 e_0' \oplus e_1} \qquad \frac{e_1 \to_1 e_1'}{v \oplus e_1 \to_1 v \oplus e_1'} \qquad \frac{}{v \oplus v' \to_1 u}\ u = v + v' \quad \begin{array}{l}\text{where } v = n \\ \quad u = n \mid \mathsf{true} \mid \mathsf{false} \\ \quad \oplus \in \{\leq, =, +, *, -\}\end{array}$$

$$\frac{e_0 \to_1 e_0'}{\langle e_0, e_1\rangle \to_1 \langle e_0', e_1\rangle} \qquad \frac{e_1 \to_1 e_1'}{\langle v, e_1\rangle \to_1 \langle v, e_1'\rangle} \qquad \frac{e_0 \to_1 e_0'}{\mathsf{fst}\ e_0 \to_1 \mathsf{fst}\ e_0'}$$

$$\frac{}{\mathsf{fst}\ \langle v, v'\rangle \to_1 v} \qquad \frac{e_0 \to_1 e_0'}{\mathsf{snd}\ e_0 \to_1 \mathsf{snd}\ e_0'} \qquad \frac{}{\mathsf{snd}\ \langle v, v'\rangle \to_1 v'}$$

$$\frac{e_0 \to_1 e_0'}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \to_1 \mathsf{if}\ x\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1} \qquad \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e_1 \to_1 e_1[x \mapsto v]} \quad \begin{array}{l}\text{where } v = n \mid \mathsf{true} \mid \mathsf{false} \\ \quad x \mid \langle v, v\rangle \mid \lambda x.\, e\end{array}$$

$$\frac{e_0 \to_1 e_0'}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to_1 \mathsf{if}\ e_0'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \qquad \frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to_1 e_1} \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to_1 e_2}$$

$$\frac{}{\mathsf{rec}\ y\ (\lambda x.\, e) \to_1 \lambda x.\, e\,[y \mapsto \mathsf{rec}\ y\ (\lambda x.\, e)]}$$

Some more intuition might be required to understand the form of the rule:

$$\frac{}{\mathsf{rec}\ y\ (\lambda x.\, e) \to_1 \lambda x.\, e\,[y \mapsto \mathsf{rec}\ y\ (\lambda x.\, e)]}$$

In Lambda Calculus, we defined recursion as the fixed point of a higher order function that takes a recursive function as its argument. If we create such a higher order function for the argument to rec then the rule follows from $\beta$-reduction and treating rec as the fixed point of the constructed higher order function.

$$\begin{array}{rcl}\text{Let } g & = & \lambda y.\, \lambda x.\, e \\ \mathsf{rec}\ y\ (\lambda x.\, e) & = & g(\mathsf{rec}\ y\ (\lambda x.\, e)) \\ & \to_\beta & \lambda x.\, e\,[y \mapsto \mathsf{rec}\ y\ (\lambda x.\, e)]\end{array}$$

[1]The semantics require that certain expressions evaluate to a certain value, v, that is different in different rules. Clearly, it is possible for the required expressions to evalutate to incorrect values in certain rules. This introduces error into the semantics. In the next lecture, we will discuss the denotational semantics of FUN, introduce domains for the various constructs, and create an error domain. With these domains and the denotational semantics, we can see how to deal with these errors, but in operational semantics, since there is not applicable rules for the error cases, the evalution of erroneous expressions just gets stuck.