CS611 Lecture 9 Reductions and Normal Form 13th September 2000

Scribe: Amy Gale, Anirban Dasgupta Lecturer: Andrew Myers

1 Reductions

The previous lecture introduced two reduction of λ-calculus, namely the α and the β reductions.

$$(\lambda x\ e) \xrightarrow{\alpha} \lambda x' e\{x'/x\} \quad x' \notin FV[e]$$

$$((\lambda x\ e_1)e_2) \xrightarrow{\beta} e_1\{x/e_2\}$$

The α-reduction corresponds to change of bound variables and the β-reduction corresponds to function application.

1.1 Extensionality

The extension of a function $f(x)$ is defined as the set $\{\langle x,\ f(x)\rangle \mid x \in \mathrm{dom}(f)\}$. Two functions are said to be equal by extension if they have the same extensions. Intuitively this means that they have a common domain of definition and they give the same result when applied to the same arguments. With lazy evaluation semantics the two functions $(\lambda x(e\ x))$ and e are equal by extension. When applied to the same argument e' the result that is obtained in both cases is $e\ e'$. This brings the concept of the third reduction in λ-calculus, the η-reduction.

$$(\lambda x(ex)) \xrightarrow{\eta} e \text{ if } x \notin FV[\![e]\!]$$

Problem However, if we insist on non-lazy evaluation, that is, the evaluation of e beforehand, then the two expressions differ if e represents a nonterminating computation. In that case, $(\lambda x(e\ x))$ represents the body of a function which goes into an infinite loop when called with any argument, while the expression e represents the divergent computation itself.

So the possible reductions on open/closed λ-terms are

$$(\lambda x\ e) \xrightarrow{\alpha} \lambda x'\ e\{x'/x\} \quad x \notin FV[\![e]\!]$$

$$((\lambda x\ e_1)e_2) \xrightarrow{\beta} e_1\{x/e_2\}$$

$$(\lambda x(e\ x)) \xrightarrow{\eta} e \quad \text{if } x \notin FV[\![e]\!]$$

A reducible expression is referred to as a **redex**. Note that we can define orders of evaluation in which reduction is allowed even for subterms of a closed λ-term, provided we have a correct redex. This brings the concept of a **normal form** for closed λ-terms.

Definition 1.1 *A λ-term is said to be in normal form if no other reductions can be performed on it or on any of its subexpressions.*

A few important points regarding a normal form are

- The normal form is defined relative to the set of reductions that we allow. The normal form could be looked upon as the value of the program represented by the λ-term.
- An expression may or may not have a normal form. In particular, expressions that represent recursive/divergent computation do not have a normal form. Y does not have a normal form.

  $$\mathrm{Y} \equiv\ \lambda f((\lambda x\ (f\ (xx)))\ (\lambda x\ (f\ (xx))))\ \xrightarrow{\beta}\ \lambda f\ f\ ((\lambda x\ (f\ xx))(\lambda x\ (f\ xx)))$$

  As we can easily see, the reduction of **Y** is never going to reach normal form, i.e. a form where no more reductions are possible.

2 Order of Reductions

2.1 Normal Order

In Normal order evaluation, the leftmost redex is evaluated first. β or η reductions are applied on the leftmost λ-subterm till no reductions can be applied. This is a form of **lazy** or **call by name** evaluation since in any application, the evaluation of the argument is delayed. Normal order has the property of being able to reduce the expression to normal form if there is one. However, Normal order is hard to implement because in passing unevaluated arguments we would have to represent computations that have not been performed, and might possibly diverge.

$$\frac{e_0 \Downarrow \{\lambda x\ e_2\}}{(e_0 e_1) \Downarrow e_2\{e_1/x\}}$$

2.2 Applicative Order

In Applicative order evaluation, before calling a function we reduce each of its arguments to the corresponding values(normal forms).

$$\frac{e_0 \rightarrow e_0'}{(e_0 e_1) \rightarrow (e_0' e_1)}$$

$$\frac{e_1 \rightarrow e_1'}{(v e_1) \rightarrow (v e_1')}$$

$$\frac{}{((\lambda x\ e) v) \rightarrow e\{v/x\}}$$

In the above expressions, v represents the value i.e. normal form of a subexpression. The Applicative order semantics is almost equivalent to **call-by-value** semantics, except that the evaluation of a function is *curried*, i.e. for a multi argument function, the arguments are evaluated one by one. The Applicative order of evaluation may end up diverging because it evaluates its arguments first. For example, the λ-term $((\lambda b\ c)((\lambda a\ aa)(\lambda a\ aa)))$ will diverge if evaluated using Applicative order semantics. But when evaluated using Normal order, the term reaches a normal form c. If the λ-calculus is to be a viable programming language, the early evaluation of the arguments would also necessitate a **non-strict IF form** in the language, because otherwise terms such as **IF TRUE 0 Y** will diverge under applicative evaluation, rather than terminating with a normal form **0**.

2.3 Nondeterministic Semantics

Since reductions of all redexes preserve meaning, we can define a Nondeterministic evaluation order for λ-terms in which we may choose to reduce any subexpression which forms a redex.

$$\frac{e_1 \rightarrow e_1'}{(e_0\ e_1) \rightarrow (e_0\ e_1')}$$

$$\frac{e_0 \rightarrow e_0'}{(e_0\ e_1) \rightarrow (e_0'\ e_1)}$$

$$\frac{e \rightarrow e'}{(\lambda x\ e) \rightarrow e'}$$

$$((\lambda x\ e_1) e_2) \xrightarrow{\beta} e_1\{x/e_2\}$$

$$(\lambda x (e\ x)) \xrightarrow{\eta} e \quad \text{if } x \notin FV[\![e]\!]$$

3 Church-Rosser Theorem

The Church-Rosser theorem assures that nondeterministic evaluation of an expression does not result in nondeterminism of the reduced normal form. Formally,

$$(e_0 \xrightarrow{*} e_1) \wedge (e_0 \xrightarrow{*} e_2) \Rightarrow \exists e_3 \; e_1 \xrightarrow{*} e_3 \wedge e_2 \xrightarrow{*} e_3' \wedge e_3 \equiv^{\alpha} e_3'$$

Any transition relation which satisfies this theorem is said to have Church-Rosser (CR) property or *diamond* property. The CR property assures the existence of only one normal form for an expression, subject to α-equivalence. CR property is true for the transition relations β and $\beta + \eta$ for any evaluation whatsoever, even for divergent computations. This suggests that we can evaluate sub-expressions of a λ-calculus term concurrently without changing the result.

4 Concurrency

In λ-calculus, the transition rules (Nondeterministic order) permit the parallel evaluation of operator and operand. The CR-property assures a deterministic result in spite of any interleaving of computation, so a concurrent evaluation of the subterms of a λ-term is possible. This is not true in the case of most imperative languages, e.g. C or C++. Consider the following statement in C : **y=(x=2) + x;** . The result stored in y entirely depends on the order in which the C-compiler decides to evaluate the two operands of **+**. In Java, however, the order of evaluation of operands of a function is completely specified, hence CR-property should have been trivially satisfied. However, the *threads* in Java do not assure a deterministic evaluation. Also, the order of initialization of the various classes may cause static constants to come up with different values, violating the CR-property. The moral of the discussion is that λ-calculus has no concept of a mutable store, and each function is evaluated locally, without side-effects. This semantics gives it the CR-property and gives implementation of functional languages more latitude in using parallelism, in principle.

5 Evaluation Contexts

We can specify the order of evaluation more compactly by defining evaluation contexts. We use the definition of contexts to define the set of legal redexes for a particular evaluation order. Given a context C and a redex e, the evaluation of a particular redex in the context takes place as

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}$$

For Normal order the legal contexts for redexes can be defined as

$$C := [\cdot] \mid Ce$$

For Applicative order, the legal contexts are

$$C := [\cdot] \mid Ce \mid \; (\lambda x \; e)C$$

Besides this we need to use a β-reduction rule which enforces function evaluation only after evaluation of the arguments.

$$\frac{}{((\lambda x \; e)v) \rightarrow e\{v/x\}}$$

And the legal contexts for Nondeterministic order are

$$C := [\cdot] \mid Ce \mid eC \mid \lambda x \; C$$

6 Simplifying λ-calculus

In simplifying λ-calculus, the variables would be the prime target. It has been shown that we can restrict, even get rid of variables by use of combinators (a fancy name for closed λ expressions). The **S** and **K** combinators eliminate the need for variables or abstractions in a closed λ-term. Any closed λ-term can be expressed as a tree of applications of **S** and **K** combinators. The **X** combinator alone is enough to express both **S** and **K** combinators by suitable applications on itself. Thus any closed λ-term could be expressed as just a tree structure, the **X** combinators on the leaves being implied.

6.1 De-Bruijn indices

In De-Bruijn form of a closed λ-term, we replace all the variable names by whole numbered indices. A variable index intuitively refers to how many λ-s one has to to walk up the abstract syntax tree in order to find this variable being bound. Some examples are

$$IDENTITY = (\lambda a\ a) = (\lambda 0)$$

$$TRUE = (\lambda x(\lambda y\ x)) = (\lambda(\lambda\ 1))$$

$$FALSE = (\lambda x(\lambda y\ y)) = (\lambda(\lambda\ 0))$$

$$2 = (\lambda f(\lambda a(f(f\ a)))) = (\lambda(\lambda(1\ (1\ 0))))$$

We could easily write the function that converts any λ-term to the corresponding De-Bruijn expression. Let **DB** $[\![e]\!]$ be the function which takes a closed λ-term and returns the De-Bruijn term. We can define **DB** in terms of a function $\mathbf{T}[\![e]\!]N$ where $N : var \rightarrow \omega$ is a map from the variable set to whole numbers denoting the numbering. Given the map **T**, the De-Bruijn transformation of a closed term is given by

$$DB[\![e]\!] = T[\![e, \emptyset]\!]N$$

Also, **T** can be calculated as

$$T[\![x]\!]N = N(x)$$

$$T[\![(e_0 e_1)]\!]N = T[\![e_0]\!]N\ T[\![e_1]\!]N$$

$$T[\![\lambda x\ e]\!]N = \lambda\ (T[\![e]\!]\lambda x' \in var.(\textbf{if}\ x' = x\ \textbf{then}\ 0\ \textbf{else}\ N(x') + 1))$$