Scribe: Ymir Vigfusson, Daria Sorokina Lecturer: Andrew Myers

1 Term equivalence

When are two terms equal to one another? Consider, for example, $\lambda x.x$ and $\lambda x.\lambda y.x$. Are these two terms equal? It turns out that defining a precise mathematical model for determining equivalence of two λ-terms is tricky. In fact Alonzo Church received the Turing-award for explaining how to define functions in Lambda-calculus. One way of attacking the problem is to try all possible arguments and compare the results. However, we cannot be sure whether we get a result or not. Let us formulate this.

We say that an expression e *terminates* or *converges* if there is a finite sequence of expressions such that

$$e \to e' \to e'' \to \cdots \to v$$

where v is a value. This is denoted by $e \Downarrow v$. Otherwise, when $e \to e' \to e'' \to \cdots$ and we never get anywhere, we say that e *diverges*. This is denoted by $e \Uparrow$.

There are infinitely many divergent terms. An example of one is Ω which was defined in the last lecture. In some way they are equal to one another, they all go off to infinity. We are now ready to give a better notion of equality.

1.1 Equality of terms

Two terms are *equal* if:

- they behave the same in every context, or
- they cause divergence in every context.

If two terms converge, we should naturally require that they converge to the same value. But we do not need this requirement explicitly, because if they converge to different values, there exists a context in which they behave differently.

In an attempt to formalize the previous definition, consider the following notation.

- $C[\cdot]$ is a context, an expression with a hole.
- $C[e]$ is C with the hole replaced by e

We then define equality in the following manner.

$$e_1 = e_2 \text{ if for all } C[\cdot] \text{ we have } C[e_1] \Downarrow \iff C[e_2] \Downarrow .$$

Sounds simple in the mathematical sense, but this is hard to compute. We need to test all different contexts, and so we would need to enumerate them all. We may also want to talk about whether open terms are equal to one another, and there we need to be more careful.

We need a conservative test of whether two terms are equal. However, we must realize that we can never tell precisely whether two terms are equal to one another, else we would have solved the halting problem. But how close can we get?

How about the following definition? Let e_1 and e_2 be expressions. If

$$e_1 \to e_1' \to \cdots \to v_1,$$
$$e_2 \to e_2' \to \cdots \to v_2,$$

and $v_1 = v_2$ then $e_1 = e_2$.

In other words, if we can show that two expressions evaluate to the same thing, then the expressions are equal. This is especially useful for compiler optimization. The previous definition raises the following questions.

- What should be the right notion of equality in this sense?
- What kind of rewrite rules (i.e. $e_1 \to e_2$) should we allow?

2 Rewrite rules

The order of the following rewrite rules is somewhat arbitrary.

2.1 β-reduction

A *β-reduction* is the following rule:

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

For example,

$$(\lambda x.\underbrace{(\lambda y.y)x}_{\beta \text{ redex}}) \xrightarrow{\beta} \lambda x.x$$

2.2 α-equivalence

In $\lambda x.xz$ the name of the bound variable x doesn't really matter. This term should really be the same as e.g. $\lambda y.yz$. Renamings like that are known as *α-conversions* or *α-renamings*. In other words: $\lambda x.xz$ and $\lambda y.yz$ are *α-equivalent* to one another. Note that this defines an equivalence relation on the set of terms, so $e_1 =_\alpha e_2$ is well defined.

Let's introduce the notation of $FV(e)$ as the set of free variables of e. In general we have

$$\lambda x.e =_\alpha \lambda x'.e\{x'/x\}.$$

But what if x' is already being used in e? Then instead we must define

$$\lambda x.e =_\alpha \lambda x'.e\{x'/x\} \text{ if } x' \notin FV(e).$$

When writing a λ-interpreter, the job of looking for α-renamings doesn't seem all that practical. However, we can e.g. use them to improve our earlier definition of equality:

If

$$e_1 \to e_1' \to \cdots \to v_1, \text{ and}$$
$$e_2 \to e_2' \to \cdots \to v_2,$$

and $v_1 =_\alpha v_2$ then $e_1 = e_2$.

Figure 1: Stoy-diagram for $\lambda x.(\lambda y.y)x$

We can create a *Stoy diagram* for a closed term in the following manner. Instead of writing $\lambda x.(\lambda y.y)x$, we write $\lambda \cdot .(\lambda \cdot .\cdot)\cdot$ and connect variables that are the same by edges. Then α-equivalent terms have the same Stoy-diagram.

2.3 η-reduction

Think about e and $\lambda x.ex$. If these two things represent functions, then you apply them to something, say e', what's going to happen? They are the same, because ee' and $(\lambda x.ex)e' \xrightarrow{\beta} ee'$, so they β-reduce to the same thing. We must be sure that x does not appear inside of e.

Formally:

$$(\lambda x.e_1x)e_2 \xrightarrow{\beta} e_1e_2 \text{ if } x \notin FV(e_1)$$

This we call an *η-reduction*, that is

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

In other words, if we apply two expressions to the same thing, they reduce to the same thing.

This reverse operation, i.e. an *η-expansion*, is practical as well. It was e.g. used to create Ω in the last lecture.

But there is a problem, what if e diverges? We need another condition:

$$e \Uparrow \iff \lambda x.ex \Uparrow$$

3 Normal forms

What if the rewrite rules do more than one thing? If we do not fix the ordering in which rewrite rules should be done, such as in CBV, you can have a choice between rewrite rules.

For example, consider that

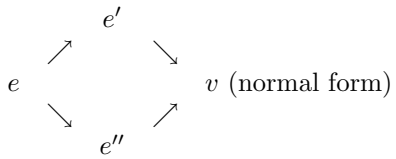
$$e_1 \to e_1' \to \cdots \to v_1$$

and

$$e_2 \to e_2' \to \cdots \to v_2.$$

But what if you can have $e_1' \to v_3$? Then e_1 and e_2 are not necessarily equal. This is not a problem, because under β-reduction or $\beta + \eta$ reduction rewriting is *confluent*. This is also known as the *diamond property* or *Church-Rosser* property.

What is confluence? The Church-Rosser theorem says that the λ-calculus that regardless of order of rewrite rules, we will arrive at a unique final result if one exists. This final form is called the *normal form*. However, because it is possible for an expression not to terminate, it does not necessarily have a normal form.



Note that under Call By Value, there is at most one rule possible at each time, so you will not get the above diamond form.

It may not be apparent, but there is a catch. We may not always pick the *easiest* rewrite rule to expedite termination, in fact, we may get stuck in a loop or a *trace* when rewriting. However, the Church-Rosser theorem provides a guarantee that it is possible to get unstuck, provided that the end result exists. Let's formalize this. We will denote by an arrow closure, that is $e_1 \to^* e_2$, that we perform zero or more rewrite rules to get from e_1 to e_2.

Suppose $e_1 \to^* e_2$ and $e_1 \to^* e_2'$. More formally, the Church-Rosser theorem states that there exists an expression e_3 such that $e_2 \to^* e_3$ and $e_2' \to^* e_3$. This tells us that if we hit a trace when rewriting a reducible expression (we get stuck), we can always get unstuck and hit the value. (A value is something you cannot do any more rewrites on, except for α-reductions)

However, we could also get stuck by doing an infinite number of rewrites. For example:

$$(\lambda xy.y)\Omega \xrightarrow{\beta} \lambda y.y$$

Yet in this case, if we choose evaluate the right-most argument, we get

$$(\lambda xy.y)\Omega \xrightarrow{\beta} (\lambda xy.y)\Omega \to_\beta \cdots$$

and we're fine. There is an ordering to prevent such traces, a so-called *normal order* where you reduce the *leftmost* redex. By using normal order, we are guaranteed to get to a normal form if one exists. This corresponds to Call By Name evaluation. Another ordering is the *applicative order* where one reduces arguments

to the normal form, and then applies β-reductions. This corresponds to Call By Value evaluation.

Example: In C the evaluation order of arguments to functions is implementation-specific. Also, C does not specify in what order operands work. Hence C is not confluent.

To demonstrate this, in C, $(x = 1) + x$ can be evaluated as 2 if you start by evaluating the left argument to the + operand. If you start with the right argument, it evaluates to $x + 1$. This depends on the implementation.