1 Term Equivalence

When are two terms equal? This is not as simple a question as it may seem. As *intensional* objects, two terms are equal if they are identical. As *extensional* objects, however, two terms should be equal if they represent the same function. But this is not as simple as it sounds, and in fact defining a precise mathematical model for defining equivalence of two λ-terms is far from straightforward.

One way of approaching the problem is to try all possible arguments and compare the results. However, one immediate problem is to know what the valid arguments are. We also need to say what happens when one or both of the computations can loop infinitely (diverge).

For starters, let us assume that we are working with an evaluation strategy such as CBV or CBN that is *deterministic*, which means that there is at most one next β-reduction that can be performed. We say that a term e *terminates* or *converges* if there is a finite sequence of reductions

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

where v is a value. We write $e \Downarrow v$ when this happens, and we write $e \Downarrow$ when $e \Downarrow v$ for some v. The other possibility is that it keeps on reducing forever without ever arriving at a value. When this happens, we say that e *diverges* and write $e \Uparrow$.

With CBN or CBV, there are infinitely many divergent terms. One example is Ω which was defined in the last lecture. In some sense, all divergent terms are equal to one another, since none of them produce a value.

We are now ready to give a better notion of equality.

1.1 Equality of Terms

Intuitively, two terms will be considered equal if in every context, either

- they both converge and produce the same value, or
- they both diverge.

A *context* is just a term $C[\,\cdot\,]$ with a single occurrence of a distinguished special variable, called the *hole*, and $C[e]$ denotes the context $C[\,\cdot\,]$ with the hold replaced by the term e. Here we do not do safe substitution, but just stuff e into the hole with abandon, allowing free variables to be captured. Then we then define equality in the following way:

$$e_1 = e_2 \quad\Longleftrightarrow\quad \text{for all contexts } C[\,\cdot\,],\ C[e_1] \Downarrow v \text{ iff } C[e_2] \Downarrow v.$$

It turns out that without loss of generality, we can simplify the definition to

$$e_1 = e_2 \quad\Longleftrightarrow\quad \text{for all contexts } C[\,\cdot\,],\ C[e_1] \Downarrow \text{ iff } C[e_2] \Downarrow,$$

because if they converge to different values, it is possible to devise a context that causes one to converge and the other to diverge.

This may sound simple in the mathematical sense, but of course it is undecidable to determine whether two terms are equal. Given the relationship between the λ-calculus and Turing machines, if we could decide equality, then we would have solved the halting problem.

A conservative approximation (but unfortunately still undecidable) is the following. Let e_1 and e_2 be terms, and suppose that e_1 and e_2 converge to the same value. Then $e_1 = e_2$. This is especially useful for compiler optimization.

2 Rewrite Rules

2.1 Recap—β-reduction

Recall that *β-reduction* is the following rule:

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

An instance of the left-hand side is called a *redex* and the corresponding instance of the right-hand side is called the *contractum*. For example,

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

Note that in CBV, $\lambda x.(\lambda y.y)x$ is a value (we cannot apply a β-reduction inside the body of an abstraction) so we cannot apply this reduction.

2.2 α-reduction

In $\lambda x.xz$ the name of the bound variable x doesn't really matter. This term is semantically the same as $\lambda y.yz$. Renamings like that are known as *α-reductions*. In an α-reduction, the new bound variable must be chosen so as to avoid capture. If a term α-reduces to another term, then the two terms are said to be *α-equivalent*. This defines an equivalence relation on the set of terms, denoted $e_1 =_\alpha e_2$.

Recall the definition of free variables $FV(e)$ of a term e. In general we have

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

The proviso $y \notin FV(e)$ is to avoid the capture of a free occurrences of y in e as a result of the renaming.

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

$$\text{If } e_1 \Downarrow v_1,\ e_2 \Downarrow v_2, \text{ and } v_1 =_\alpha v_2, \text{ then } e_1 = e_2.$$

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

Here is another notion of equality. Compare the terms e and $\lambda x.\, ex$. If these two terms are both applied to an argument e', then they will both reduce to $e\,e'$, provided x has no free occurrence in e. Formally,

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

This says that e and $\lambda x.\, ex$ behave the same way as functions and should be considered equal. Another way of stating this is that e and $\lambda x.\, ex$ behave the same way in all contexts of the form $[\,\cdot\,]\,e'$.

This gives rise to a reduction rule called *η-reduction*:

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

The reverse operation, called *η-expansion*, is practical as well.

In practice, η-expansion is used to delay divergence by trapping expressions inside λ terms.

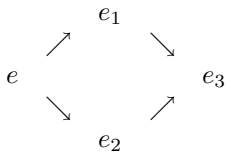
The η rule may not be sound with respect to our earlier notion of equality, depending on our reduction strategy. For example, $\lambda x.\, ex$ is a value in CBV, but reductions might be possible in e and it might diverge.

3 The Church–Rosser Property

In the classical λ-calculus, no reduction strategy is specified, and no restrictions are placed on the order of reductions. Any redex may be chosen to be reduced next. A λ-term in general may have many redexes, so the process is nondeterministic. We can think of a reduction strategy as a mechanism for resolving the nondeterminism, but in the classical λ-calculus, no such strategy is specified. A *value* in this case is just a term containing no redexes. Such a term is said to be in *normal form*.

This makes it very difficult to define equality. One sequence of reductions may terminate, but another may not. It is even conceivable that different terminating reduction sequences result in different values. Luckily, it turns out that the latter cannot happen.

It turns out that the λ-calculus is *confluent* (also known as the *Church–Rosser* property) under α- and β-reductions. Confluence says that if e reduces by some sequence of reductions to e_1, and if e also reduces by some other sequence of reductions to e_2, then there exists an e_3 such that both e_1 and e_2 reduce to e_3.



It follows that up to α-equivalence, normal forms are unique. For if $e \Downarrow v_1$ and $e \Downarrow v_2$, and if v_1 and v_2 are in normal form, then by confluence they must be α-equivalent. Moreover, regardless of the order of previous reductions, it is always possible to get to the unique normal form if it exists.

However, note that it is still possible for a reduction sequence not to terminate, even if the term has a normal form. For example, $(\lambda x.\, \lambda y.\, y)\Omega$ has a nonterminating CBV reduction sequence

$$(\lambda xy.y)\Omega \quad \stackrel{\beta}{\longrightarrow} \quad (\lambda xy.y)\Omega \quad \stackrel{\beta}{\longrightarrow} \quad \cdots$$

but a terminating CBV reduction sequence, namely

$$(\lambda x.\, \lambda y.\, y)\Omega \quad \stackrel{\beta}{\longrightarrow} \quad \lambda y.\, y.$$

It may be difficult to determine the most efficient way to expedite termination. But even if we get stuck in a loop, the Church–Rosser theorem guarantees that it is always possible to get unstuck, provided the normal form exists.

In call-by-name (CBN), the leftmost redex is always reduced first. This strategy is also called *normal order*. It turns out that CBN is guaranteed to find the normal form if it exists, albeit not necessarily in the most efficient way. Call-by-value (CBV) is also called *applicative order*.

In C, the order of evaluation of arguments is implementation-specific. Also, C does not specify in what order operands work. Hence C is not confluent. For example, the value of the expression $(x = 1) + x$ is 2 if the left operand of $+$ is evaluated first, $x+1$ if the right operand is evaluated first. The language specification does not say which, so it depends on the implementation.