Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 Church Encodings

See Pierce, Section 5.2 for encodings of Church booleans, pairs, and Church numerals.

We can encode boolean values and conditionals in the λ-calculus as follows:

$$\begin{array}{lcl} \mathsf{tru} & = & \lambda x.\, \lambda y.\, x \\ \mathsf{fls} & = & \lambda x.\, \lambda y.\, y \\ \mathsf{ife} & = & \lambda b.\, \lambda m.\, \lambda n.\, b\ m\ n \end{array}$$

To see how the encoding works, try reducing the following expression using CBV:

$$\mathsf{ife}\ \mathsf{tru}\ v_1\ v_2.$$

Next, consider the expression:

$$\mathsf{ife}\ \mathsf{tru}\ \mathsf{id}\ \Omega$$

Notice that under CBV, the above expression loops forever, while under CBN, it evaluates to id.

The general trick for doing these encodings is as follows. Values will be functions; construct these functions so that they return the appropriate information when called by an operation.

2 Lambda Calculus with Booleans

We can also directly extend the λ-calculus with booleans and conditionals, rather than rely on encodings. The syntax of the extended language λ+bool is as follows:

$$e \quad ::= \quad x \ \mid\ \lambda x.\, e \ \mid\ e_0\ e_1 \ \mid\ \mathsf{true} \ \mid\ \mathsf{false} \ \mid\ \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

Before we give a CBV semantics for the extended language, we must define the set of values in the language:

$$v \quad ::= \quad \lambda x.\, e \ \mid\ \mathsf{true} \ \mid\ \mathsf{false}$$

Let us now define a CBV structural operational semantics for λ+bool. There is more than one way to evaluate conditionals in this language. For now, we adopt the following operational semantics. We write $\longrightarrow_a$ for the small-step relation. (Later we will consider a different small-step relation $\longrightarrow_b$ for λ+bool.)

$$\frac{e_1 \longrightarrow_a e_1'}{e_1\ e_2 \longrightarrow_a e_1'\ e_2} \qquad \frac{e \longrightarrow_a e'}{v\ e \longrightarrow_a v\ e'} \qquad \frac{}{\lambda x.\, e\ v \longrightarrow_a e\{v/x\}}$$

$$\frac{e_0 \longrightarrow_a e_0'}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow_a \mathsf{if}\ e_0'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2} \qquad \frac{e_1 \longrightarrow_a e_1'}{\mathsf{if}\ v_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow_a \mathsf{if}\ v_0\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2}$$

$$\frac{e_2 \longrightarrow_a e_2'}{\mathsf{if}\ v_0\ \mathsf{then}\ v_1\ \mathsf{else}\ e_2 \longrightarrow_a \mathsf{if}\ v_0\ \mathsf{then}\ v_1\ \mathsf{else}\ e_2'} \qquad \frac{}{\mathsf{if\ true\ then}\ v_1\ \mathsf{else}\ v_2 \longrightarrow_a v_1} \qquad \frac{}{\mathsf{if\ false\ then}\ v_1\ \mathsf{else}\ v_2 \longrightarrow_a v_2}$$

2.1 Translating the λ+bool-calculus to λ-calculus

Pierce (Chapter 5): "In the 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus capturing the language's essential mechanisms, together with a collection of convenient *derived forms* whose behavior is understood by translating them into the core ..."

As a simple example of translating a more complex programming language into a core calculus, let us consider a translation from the language λ+bool to the λ-calculus. We define a function $\mathcal{T}[\![e]\!]$ that maps a λ+bool (source language) expression e to a λ-calculus (target language) term. The translation of variables, λ abstraction, and application is straightforward since λ-abstraction and application are available in the target language. When translating boolean values and conditionals, we use the Church encoding of booleans that we discussed above.

$$\begin{array}{rcl} \mathsf{tru} & = & \lambda x.\, \lambda y.\, x \\ \mathsf{fls} & = & \lambda x.\, \lambda y.\, y \\ \mathsf{ife} & = & \lambda b.\, \lambda m.\, \lambda n.\, b\ m\ n \end{array}$$

$$\begin{array}{rcl} \mathcal{T}[\![x]\!] & = & x \\ \mathcal{T}[\![\lambda x.\, e]\!] & = & \lambda x.\, \mathcal{T}[\![e]\!] \\ \mathcal{T}[\![e_1\ e_2]\!] & = & (\mathcal{T}[\![e_1]\!])\ (\mathcal{T}[\![e_2]\!]) \\ \mathcal{T}[\![\mathsf{true}]\!] & = & \mathsf{tru} \\ \mathcal{T}[\![\mathsf{false}]\!] & = & \mathsf{fls} \\ \mathcal{T}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & = & \mathsf{ife}\ \mathcal{T}[\![e_0]\!]\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!]] \end{array}$$

Though the above translation function is recursive, notice that in each case, the right-hand side defines the value of $\mathcal{T}[\![e]\!]$ in terms of proper subterms of e. Since all terms have finite size, this means that if we were to expand out the definition of $\mathcal{T}[\![e]\!]$ for any given e, it would eventually bottom out at applications to variables, or at applications to true or false. Technically, this is a definition of a function by *structural induction* on e.

You can also think of $\mathcal{T}[\![e]\!]$ as a *compiler* that compiles the source term e into a target term. An important feature of such a translation (or of a compiler) is that the translation *preserve* the semantics of the source language. (Such compilers are called *semantics preserving*.) Intuitively, the behavior of the target term $\mathcal{T}[\![e]\!]$ should match the behavior of the source term e. Specifically, the translation is *sound* if the following holds: If $e \longrightarrow_a v$, then there exists v' such that $\mathcal{T}[\![e]\!] \longrightarrow v'$ and $\mathcal{T}[\![v]\!] \equiv v'$. (We will define equivalence $\equiv$ later in this lecture. For now, you can think of $\equiv$ as equality.)

Given the source-language operational semantics ($\longrightarrow_a$) the function $\mathcal{T}[\![e]\!]$ is sound. Later on we will see how to prove such properties.

In particular, note that the source term $\mathsf{if\,true\,then\,id\,else}\,\Omega$ diverges (when reduced using $\longrightarrow_a$), and that its translation $\mathcal{T}[\![\mathsf{if\ true\ then\ id\ else}\ \Omega]\!] = \mathsf{ife\ tru\ id}\ \Omega$ also diverges (when reduced using the CBV operational semantics $\longrightarrow$).

2.2 An alternative operational semantics for λ+bool

The operational semantics we gave above for λ+bool is rather unusual in that both branches of the conditional are always evaluated. In real programming languages, only one branch is evaluated depending on whether the condition evaluates to true or false. Below we define such an operational semantics; we write $\longrightarrow_b$ for this small-step relation.

$$\frac{e_1 \longrightarrow_b e_1'}{e_1\ e_2 \longrightarrow_b e_1'\ e_2} \qquad \frac{e \longrightarrow_b e'}{v\ e \longrightarrow_b v\ e'} \qquad \frac{}{\lambda x.\, e\ v \longrightarrow_b e\{v/x\}}$$

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

If we use the translation function we defined above, notice that the translation is no longer *sound*. In particular, note that the source term $\mathsf{if\ true\ then\ id\ else}\ \Omega \longrightarrow_b \mathsf{id}$, whereas its translation $\mathcal{T}[\![\mathsf{if\ true\ then\ id\ else}\ \Omega]\!] = \mathsf{ife\ tru\ id}\ \Omega$ diverges (under the CBV operational semantics $\longrightarrow$).

How can we fix the translation so that it is sound when the source language operational semantics is given by $\longrightarrow_b$? In particular, we want a translation $\mathcal{T}[\![e]\!]$ such that: If $e \longrightarrow_b v$, then there exists v' such that $\mathcal{T}[\![e]\!] \longrightarrow v'$ and $\mathcal{T}[\![v]\!] \equiv v'$.

The key idea is to change the translation of if expressions such as $\mathsf{if\,true\,then\,id\,else}\,\Omega$ so that the evaluation of the Ω is delayed until it is needed. That is, the translation should put the diverging computation (in this

case, Ω) under a λ, as in $\lambda z.\,\Omega$. Since λ-abstractions are values, the term $\lambda z.\,\Omega$ will not lead to an infinite loop unless it is applied. Specifically, we can change the translation of if expressions as follows:

$$\mathcal{T}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] \quad = \quad \mathsf{ife}\ \mathcal{T}[\![e_0]\!]\ (\lambda y.\,\mathcal{T}[\![e_1]\!])\ (\lambda z.\,\mathcal{T}[\![e_2]\!]) \quad \text{where } y \notin FV(e_1) \text{ and } z \notin FV(e_2)$$

However, the above is now no longer in sync with the encodings of tru and fls chosen above. We change the encodings of tru and fls and the translation as follows.

$$\begin{aligned}
\mathsf{tru}' &= \lambda x.\,\lambda y.\,x\ \mathsf{id} \\
\mathsf{fls}' &= \lambda x.\,\lambda y.\,y\ \mathsf{id} \\
\mathsf{ife}' &= \lambda b.\,\lambda m.\,\lambda n.\,b\ m\ n
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x.\,e]\!] &= \lambda x.\,\mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= (\mathcal{T}[\![e_1]\!])\ (\mathcal{T}[\![e_2]\!]) \\
\mathcal{T}[\![\mathsf{true}]\!] &= \mathsf{tru}' \\
\mathcal{T}[\![\mathsf{false}]\!] &= \mathsf{fls}' \\
\mathcal{T}[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] &= \mathsf{ife}'\ \mathcal{T}[\![e_0]\!]\ (\lambda y.\,\mathcal{T}[\![e_1]\!])\ (\lambda z.\,\mathcal{T}[\![e_2]\!]) \quad \text{where } y \notin FV(e_1) \text{ and } z \notin FV(e_2)
\end{aligned}$$

The translation defined above is sound. Note that the source term $\mathsf{if\ true\ then\ id\ else}\ \Omega \longrightarrow_b \mathsf{id}$, and its translation also evaluates to id (under CBV):

$$\begin{aligned}
& \mathcal{T}[\![\mathsf{if\ true\ then\ id\ else}\ \Omega]\!] \\
= \ & \mathsf{ife}'\ \mathsf{tru}'\ (\lambda z.\,\mathsf{id})\ (\lambda z.\,\Omega) \\
\longrightarrow \ & \mathsf{tru}'\ (\lambda y.\,\mathsf{id})\ (\lambda z.\,\Omega) \\
\longrightarrow \ & (\lambda y.\,\mathsf{id})\ \mathsf{id} \\
\longrightarrow \ & \mathsf{id}
\end{aligned}$$

3 Formal Treatment of Substitution

3.1 Variable Capture

CBN and CBV evaluation reduce β redexes $(\lambda x.\,e)\ e'$ only when the right-hand side (RHS) e' is a closed term. In general we can try to *normalize* lambda terms by reducing redexes *inside* lambda abstractions, because the reductions should still preserve the equivalence of the terms. However, in this case the term e' may be an open term containing free variables. Substituting e' into e may cause variable capture. For example, consider the substitution $(y\ (\lambda x.\,x\ y))\{x/y\}$. If we replace all the unbound y's with x, we'll get $x\ (\lambda x.\,x\ x)$.

In fact, this is a problem seen in many other mathematical contexts, such as in integral calculus, because like λ, the integral operator is a binder.

3.2 Free variables

In order to define substitution correctly, we first need to define the set of free variables of a term. We denote this by the function $FV(e)$, which is defined recursively as follows:

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

This definition is recursive, but notice that in each of the three cases, the right-hand side defines the value of $FV(e)$ in terms of proper subterms of e. Since all terms have finite size, this means that if we were to expand out the definition of $FV(e)$ for any given e, it would eventually bottom out at applications to variables (the first case). Technically, this is a definition of a function by *structural induction* on e. We will talk more about these kinds of definitions later.

3.3 Capture-avoiding substitution

We write $e_1\{e_2/x\}$ to denote the result of substituting e_2 for all free occurrences of x in e_1, according to the following rules, also by structural induction on e (modulo α-equivalence, which well talk about shortly):

$$
\begin{array}{lcll}
x\{e/x\} & = & e & \\
y\{e/x\} & = & y & \text{where } y \neq x \\
(e_0\ e_1)\{e/x\} & = & (e_0\{e/x\})(e_1\{e/x\}) & \\
(\lambda x.\, e_0)\{e/x\} & = & \lambda x.\, e_0 & \\
(\lambda y.\, e_0)\{e/x\} & = & \lambda y.\, e_0\{e/x\} & \text{where } y \neq x \text{ and } y \notin FV(e_1) \\
(\lambda y.\, e_0)\{e/x\} & = & \lambda z.\, e_0\{z/y\}\{e/x\} & \text{where } y \neq x,\ z \neq x,\ z \notin FV(e_0) \text{ and } z \notin FV(e).
\end{array}
$$

Note that the rules are applied inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms. The very last of the six rules applies when $y \in FV(e_1)$. In this case we can rename the bound variable y to z to avoid capture of the free occurrence of y. One might well ask: But what if y occurs free in the scope of a λz in e_1? Wouldn't the z then be captured? The answer is that it will be taken care of in the same way, but inductively on a smaller term.

Despite the importance of substitution, it was not until the mid-1950s that a completely satisfactory definition of substitution was given, by Haskell Curry. Previous mathematicians, from Newton to Hilbert to Church, worked with incomplete or incorrect definitions. It is the last of the rules above that is hard to get right, because it is easy to forget one of the three restrictions on the choice of y', or to falsely convince yourself that they are not needed.

In the pure λ-calculus, we can start with a λ-term and perform β-reductions on subterms in any order, using the full substitution rule to avoid variable capture when the substituted term is open.

Equivalence, Reduction and Normal Forms

4 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 syntactically identical. As *extensional* objects, however, two terms should be equal if they represent the same function. We will say that two terms are *equivalent* if they are equal in an extensional sense.

For example, it seems clear that the terms $\lambda x.\, x$ and $\lambda y.\, y$ are equivalent. The name of the variable is not essential. But we also probably think that $\lambda x.\, (\lambda y.\, y)\ x$ is equivalent to $\lambda x.\, x$ too, in a less trivial sense. And there are even more interesting cases, like $\lambda x.\, yx\ y$.

But what function does a term like $\lambda x.\, x$ represent? Intuitively, its the identity function, but over what domain and codomain? We might think of it as representing the set of all identity functions, but this interpretation quickly leads to Russells paradox. In fact, defining a precise mathematical model for lambda-calculus terms is far from straightforward, requiring some sophisticated domain theory.

One possible meaning of a term is divergence. There are innitely many divergent terms; one example is Ω. In some sense, all divergent terms are equivalent, since none of them produce a value. The implication is that it is undecidable to determine whether two terms are equivalent, because otherwise, given the relationship between the λ-calculus and Turing machines, we could solve the halting problem on lambda calculus terms by testing equivalence to Ω.

4.1 Observational Equivalence

Another way of approaching the problem is to say that two terms are equivalent if they behave indistinguishably in all possible contexts.

More precisely, two terms will be considered equivalent if in every context, either

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

A *program context* (usually *context* for short) is just a term with a single occurrence of a distinguished special variable called the *hole*. We write $[\cdot]$ to denote the hole. We use the metavariable C for contexts.

For the λ-calculus, a context C can be

- a hole $[\cdot]$;
- a lambda term $\lambda x.\, C$, which has a hole somewhere in the body of the λ (or *under* the λ);
- an application $C\ e$, where a term with a hole is applied to an argument without a hole; or
- an application $e\ C$, where a term without a hole is applied to an argument with a hole.

In BNF notation, here is the grammar for expressions e and contexts C for the λ-calculus.

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.\, e \mid e_0\ e_1 \\ C & ::= & [\cdot] \mid \lambda x.\, C \mid C_0\ e_1 \mid e_0\ C_1 \end{array}$$

The notation $C[e]$ denotes the context C with the hole replaced by the term e. Then we can define equivalence in the following way:

$$e_1 \equiv e_2 \iff \text{for all contexts } C,\ C[e_1] \Downarrow v \text{ iff } C[e_2] \Downarrow v.$$

Without loss of generality, we can simplify the definition to

$$e_1 \equiv e_2 \iff \text{for all contexts } C,\ C[e_1] \Downarrow \text{ iff } C[e_2] \Downarrow$$

because if they converge to different values, it is possible to devise a context that causes one to converge and the other to diverge. Suppose that $C[e_1] \Downarrow v_1$ and $C[e_2] \Downarrow v_2$, where v_1 and v_2 have different behavior. Then we can find some context C' which applied to v_1 converges, and applied to v_2 diverges. Therefore, the context $C'[C[\cdot]]$ is a context that causes the original e_1, e_2 to converge and diverge respectively, satisfying the simpler denition.

This notion of equivalence is known as *observational equivalence*, or alternatively, *contextual equivalence*. It may be written $e_1 \equiv e_2$, or $e_1 \approx^{obs} e_2$, or $e_1 \approx^{ctx} e_2$.

As mentioned above, determining whether two terms are equivalent is undecidable. 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 v when reductions are applied according to some strategy. Then e_1 is equivalent to e_2. That is,

$$\text{If } e_1 \Downarrow v \text{ and } e_2 \Downarrow v, \quad \text{then } e_1 \equiv e_2$$

This *normalization* approach (in which terms are reduced to a *normal form* on which no more reductions can be done) is useful for compiler optimization and for checking type equality in some advanced type systems.

5 Rewrite Rules

5.1 β-reduction

We have already seen the β-reduction rule:

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

An instance of the left-hand side (LHS) is called a *redex* and the corresponding instance of the right-hand side is called the *contractum*. (Note that in CBV, $\lambda x.\, (\lambda y.\, y)\ x$ is a value—we cannot apply β-reduction inside the body of an abstraction—so we cannot apply this reduction.)

If a term β-reduces to another term then the two terms are said to be *β-equivalent*. This defines an equivalence relation on the set of terms denoted $e_1 =_\beta e_2$. Hence, we have the following β-equivalence:

$$(\lambda x.\, e)\ e_1 \quad =_\beta \quad e\{e_1/x\}.$$

A term is in *β-normal form* if no β-reduction is possible.

5.2 α-reduction

In $\lambda x.\, x\ z$ the name of the bound variable x doesn't really matter. This term is semantically the same as $\lambda y.\, y\ z$. A renaming like this is known as an α-reduction, α-conversion, or just α-renaming. 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 $e1 =_\alpha e_2$.

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

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

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 (normalization-based) definition of equivalence as follows:

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

5.3 Stoy diagrams and de Bruijn indices

We can create a *Stoy diagram* (after Joseph Stoy) for a closed term in the following manner. Instead of writing a term with variable names, we write dots to represent the variables and connect variables with the same binding with edges. Then α-equivalent terms have the same Stoy-diagram. For example, the term $\lambda x.\, (\lambda y.\, (\lambda x.\, x\ y)\ x)\ x$ has the following Stoy diagram:

Another way to formalize the lambda calculus is to define its terms as the *equivalence classes* of the syntactic terms given, with respect to the α-equivalence relation. (The equivalence classes of a set are just the subsets that are equivalent with respect to some equivalence relation.) Since all terms in the equivalence class have the same Stoy diagram, we can understand the terms as Stoy diagrams, and the reductions as operating on these diagrams. This approach is often taken in theoretical programming language work, even when the presentation appears to be using explicit variable names.

A related approach is to represent variables using *de Bruijn indices*, which replace variables with natural numbers that indicate the binding site. Specifically, this is done by replacing named variables by natural numbers where the number k stands for "the variable bound by the k'th enclosing λ." For example, the term $\lambda x.\, x$ corresponds to the *nameless term* $\lambda.\, 0$, while the term $\lambda x.\, \lambda y.\, x\ (y\ x)$ corresponds to $\lambda.\, \lambda.\, 1\ (0\ 1)$. Such terms are called *de Bruijn terms* and the numeric variables in them are called *de Bruijn indices*.

5.4 η-reduction

Here is another notion of equivalence. Compare the terms e and $\lambda x.\, e\ x$. If these two terms are both applied to an argument e_1, then they will both reduce to $e\ e_1$, provided x has no free occurrence in e. Formally,

$$(\lambda x.\, e\ x)\ e_1 \quad \xrightarrow{\beta} \quad e\ e_1 \qquad \text{if } x \notin FV(e).$$

This says that e and $\lambda x.\, e\ x$ behave the same way as functions and should be considered equal. Another way of stating this is that e and $\lambda x.\, e\ x$ behave the same way in all contexts of the form $[\cdot]\ e_1$.

This gives rise to the rule for η-reductions:

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

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

The reverse operation, called η-expansion, can be useful as well. In practice, η-expansion is used to delay divergence by trapping expressions inside λ abstraction terms.

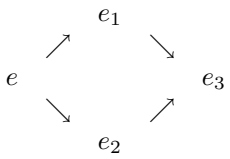
A term is in *$\beta\eta$-normal form* if neither a β-reduction nor an η-reduction is possible.

6 Confluence

In the classical λ-calculus, no reduction strategy is specied, 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 specied. 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 equivalence. 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

$$(\lambda x.\, \lambda y.\, y)\ \Omega \quad \xrightarrow{\beta} \quad (\lambda x.\, \lambda y.\, y)\ \Omega \quad \xrightarrow{\beta} \quad \cdots$$

but a terminating CBN reduction, namely

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

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 *normal order*, the leftmost redex is always reduced first. This strategy is closely related to CBN evaluation, but also reduces inside lambdas. Like CBN, it finds a normal form if one exists, albeit not necessarily in the most efficient way. Call-by-value (CBV) is correspondingly related to *applicative order*, where arguments are reduced first.

In C, the order of evaluation of arguments is not defined by the language; it is implementation-specific. Because of this, and the fact that C has side effects, C is not confluent. For example, the value of the expression $(x = 1) + x$ is 2 if the left operand of $+$ is evaluated first, and $1 + x$ if the right operand is evaluated first. This makes writing correct C programs more challenging!

The absence of confluence in concurrent imperative languages is why concurrent programming is difficult. In the lambda calculus, confluence guarantees that reduction can be done in parallel without fear of changing the result.