Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Notes on Lambda Calculus

Spring 2017

The lambda calculus (or λ-calculus) was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s to describe functions in an unambiguous and compact manner. Many real languages are directly based on the lambda calculus, such as Lisp, Scheme, Haskell, and ML. A key characteristic of these languages is that functions are values, just like integers and booleans are values: functions can be used as arguments to functions, and can be returned from functions. Those concepts have further made it into many other languages, including Python and JavaScript.

The name "lambda calculus" comes from the use of the Greek letter lambda (λ) in function definitions. (The letter lambda has no significance.) "Calculus" means a method of calculating by the symbolic manipulation of expressions.

Intuitively, a function is a rule for determining a value from an argument. Some examples of functions in mathematics are

$$\begin{aligned} f(x) &= x^3 \\ g(y) &= y^3 - 2y^2 + 5y - 6. \end{aligned}$$

(In the lambda notation introduced by Church and Kleene and used in the lambda calculus, these functions would be written $\lambda x.\, x^3$ and $\lambda y.\, y^3 - 2y^2 + 5y - 6$.)

1 Syntax

The pure λ-calculus contains just function definitions (called *abstractions*), variables, and function *application* (i.e., applying a function to an argument). If we add additional data types and operations (such as integers and addition), we have an *applied* λ-calculus. In the following text, we will sometimes assume that we have integers and addition in order to give more intuitive examples.

The syntax of the pure λ-calculus is defined as follows.

$$\begin{array}{rll} e ::= & x & \text{variable} \\ \mid & \lambda x.\, e & \text{abstraction} \\ \mid & e_1\ e_2 & \text{application} \end{array}$$

An abstraction $\lambda x.\, e$ is a function: variable x is the *argument*, and expression e is the *body* of the function. Note that the function $\lambda x.\, e$ doesn't have a name. Assuming we have integers and arithmetic operations, the expression $\lambda y.\, y \times y$ is a function that takes an argument y and returns square of y.

An application $e_1\ e_2$ requires that e_1 is (or evaluates to) a function, and then applies the function to the expression e_2. For example, $(\lambda y.\, y \times y)\ 5$ is, intuitively, equal to 25, the result of applying the squaring function $\lambda y.\, y \times y$ to 5.

Here are some examples of lambda calculus expressions.

$\lambda x.\, x$	a lambda abstraction called the *identity function*
$\lambda x.\, (f\ (g\ x)))$	another abstraction
$(\lambda x.\, x)\ 42$	an application
$\lambda y.\, \lambda x.\, x$	an abstraction that ignores its argument and returns the identity function

Lambda expressions extend as far to the right as possible. For example $\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, (x\ (\lambda y.\, y))$, and is not the same as $(\lambda x.\, x)\ (\lambda y.\, y)$. Application is left associative. For example $e_1\ e_2\ e_3$ is the same as $(e_1\ e_2)\ e_3$. In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.

1.1 Variable binding and α-equivalence

An occurrence of a variable in an expression is either *bound* or *free*. An occurrence of a variable x in a term is bound if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identifiers are bound.

Consider the following term:

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

Both occurrences of x are bound, the first occurrence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a program has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed program in lambda calculus is a closed term.

The symbol λ is a *binding operator*, as it binds a variable within some scope (i.e., some part of the expression): variable x is bound in e in the expression $\lambda x.\, e$.

The name of bound variables is not important. Consider the mathematical integrals $\int_0^7 x^2 dx$ and $\int_0^7 y^2 dy$. They describe the same integral, even though one uses variable x and the other uses variable y in their definition. The meaning of these integrals is the same: the bound variable is just a placeholder. In the same way, we can change the name of bound variables without changing the meaning of functions. Thus $\lambda x.\, x$ is the same function as $\lambda y.\, y$. Expressions e_1 and e_2 that differ only in the name of bound variables are called *α-equivalent* ("alpha equivalent"), sometimes written $e_1 =_\alpha e_2$.

1.2 Higher-order functions

In lambda calculus, functions are values: functions can take functions as arguments and return functions as results. In the pure lambda calculus, every value is a function, and every result is a function!

For example, the following function takes a function f as an argument, and applies it to the value 42.

$$\lambda f.\, f\ 42$$

This function takes an argument v and returns a function that applies its own argument (a function) to v.

$$\lambda v.\, \lambda f.\, (f\ v)$$

2 Semantics

2.1 β-equivalence

Application $(\lambda x.\, e_1)\ e_2$ applies the function $\lambda x.\, e_1$ to e_2. In some ways, we would like to regard the expression $(\lambda x.\, e_1)\ e_2$ as equivalent to the expression e_1 where every (free) occurrence of x is replaced with e_2. For example, we would like to regard $(\lambda y.\, y \times y)\ 5$ as equivalent to 5×5.

We write $e_1\{e_2/x\}$ to mean expression e_1 with all free occurrences of x replaced with e_2. There are several different notations to express this substitution, including $[x \mapsto e_2]e_1$ (used by Pierce), $[e_2/x]e_1$ (used by Mitchell), and $e_1[e_2/x]$ (used by Winskel).

Using our notation, we would like expressions $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$ to be equivalent.

We call this equivalence, between $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$, is called *β-equivalence*. Rewriting $(\lambda x.\, e_1)\ e_2$ into $e_1\{e_2/x\}$ is called a *β-reduction*. Given a lambda calculus expression, we may, in general, be able to perform β-reductions. This corresponds to executing a lambda calculus expression.

There may be more than one possible way to β-reduce an expression. Consider, for example, $(\lambda x.\, x + x)\ ((\lambda y.\, y)\ 5)$. We could use β-reduction to get either $((\lambda y.\, y)\ 5) + ((\lambda y.\, y)\ 5)$ or $(\lambda x.\, x + x)\ 5$. The order in which we perform β-reductions results in different semantics for the lambda calculus.

2.2 Evaluation strategies

There are many different evaluation strategies for the lambda calculus. The most permissive is full β-reduction, which allows any redex—i.e., any expression of the form $(\lambda x.\, e_1)\ e_2$—to step to $e_1\{e_2/x\}$ at any time. It is defined formally by the following small-step operational semantics rules.

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'} \qquad \frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

A term e is said to be in *normal form* when it cannot be reduced any further, that is, when there is no e' such that $e \longrightarrow e'$. It is convenient to say that term e *has* normal form e' if $e \longrightarrow^* e'$ with e' in normal form.

Not every term has a normal form under full β-reduction. Consider the expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, which we will refer to as Ω for brevity. Let's try evaluating Ω.

$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \longrightarrow (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) = \Omega$$

Evaluating Ω never reaches a term in normal form! It is an infinite loop!

When a term has a normal form, however, it never has more than one. This is not a given, because clearly the full β-reduction strategy is non-deterministic. Look at the term $(\lambda x.\, \lambda y.\, y)\ \Omega\ (\lambda z.\,.z)$, for example. It has two redexes in it, the one with abstraction λx, and the one inside Ω. But this nondeterminism is well behaved: when different sequences of reduction reach a normal form (they need not) those normal forms are equal.

Formally, full β-reduction is confluent in the following sense:

Theorem 1 (Confluence)**.** *If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists e' such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow^* e'$.*

Confluence can be depicted graphically as follows (where $\twoheadrightarrow$ is used to represent $\longrightarrow^*$):

$$\begin{tikzcd}
& e \arrow[dl, two heads] \arrow[dr, two heads] & \\
e_1 \arrow[dr, two heads] & & e_2 \arrow[dl, two heads] \\
& e' &
\end{tikzcd}$$

Confluence is often also called the Church-Rosser property. It is not an easy result to prove. (It would make sense for it to be a proof by induction on the multi-step reduction $\longrightarrow^*$.Try it, and see where you get stuck.)

Corollary 1. *If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ and both e_1 and e_2 are in normal form, then $e_1 = e_2$.*

Proof. An easy consequence of confluence. $\square$

Other evaluation strategies are possible, which impose a deterministic order on the reductions. For example, *normal order evaluation* uses the full β-reduction rules, except imposes the order that the left-most redex—that is, the redex in which the leading λ appears left-most in the term—is always reduced first. Normal order evaluation guarantees that if a term has a normal form, applying reductions in normal order will eventually yield that normal form.

Normal order evaluation allows reducing redexes inside abstractions, which may strike you as odd if you rely on your programmer's intuition: a function definition does not simply reduce its body, unprompted. That's because most programming languages use reduction strategies that when put in lambda calculus terms do not perform reductions inside abstractions.

Two common evaluations strategies that occur in programming languages are call-by-value and call-by-name.

Call-by-value (or CBV) evaluation strategy is more restrictive: it only allows an application to reduce after its argument has been reduced to a value and does not allow evaluation under a λ. That is, given an application $(\lambda x.\, e_1)\ e_2$, CBV semantics makes sure that e_2 is a value before calling the function.

So, what is a value? In the pure lambda calculus, any abstraction is a value. Remember, an abstraction $\lambda x.\, e$ is a function; in the pure lambda calculus, the only values are functions. In an *applied lambda calculus* with integers and arithmetic operations, values also include integers. Intuitively, a value is an expression that can not be reduced/executed/simplified any further.

We can give small-step operational semantics for call-by-value execution of the lambda calculus. Here, v can be instantiated with any value (e.g., a function).

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

We can see from these rules that, given an application $e_1\ e_2$, we first evaluate e_1 until it is a value, then we evaluate e_2 until it is a value, and then we apply the function to the value—a β-reduction.

Let's consider some examples. (These examples use an applied lambda calculus that also includes reduction rules for arithmetic expressions.)

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 \quad &\longrightarrow (\lambda x.\, \lambda y.\, y\ x)\ 7\ \lambda x.\, x+1\\ &\longrightarrow (\lambda y.\, y\ 7)\ \lambda x.\, x+1\\ &\longrightarrow (\lambda x.\, x+1)\ 7\\ &\longrightarrow 7+1\\ &\longrightarrow 8\end{aligned}$$

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) \quad &\longrightarrow (\lambda f.\, f\ 7)\ ((\lambda y.\, y)\ (\lambda y.\, y))\\ &\longrightarrow (\lambda f.\, f\ 7)\ (\lambda y.\, y)\\ &\longrightarrow (\lambda y.\, y)\ 7\\ &\longrightarrow 7\end{aligned}$$

Call-by-name (or CBN) semantics are more permissive that CBV, but less permissive than full β-reduction. CBN semantics applies the function as soon as possible. The small-step operational semantics are a little simpler, as they do not need to ensure that the expression to which a function is applied is a value.

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

Let's consider the same examples we used for CBV.

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 \quad &\longrightarrow (\lambda y.\, y\ (5+2))\ \lambda x.\, x+1\\ &\longrightarrow (\lambda x.\, x+1)\ (5+2)\\ &\longrightarrow (5+2)+1\\ &\longrightarrow 7+1\\ &\longrightarrow 8\end{aligned}$$

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) \quad &\longrightarrow ((\lambda x.\, x\ x)\ \lambda y.\, y)\ 7\\ &\longrightarrow ((\lambda y.\, y)\ (\lambda y.\, y))\ 7\\ &\longrightarrow (\lambda y.\, y)\ 7\\ &\longrightarrow 7\end{aligned}$$

Note that the answers are the same, but the order of evaluation is different. (Later we will see languages where the order of evaluation is important, and may result in different answers.)

One way in which CBV and CBN differ is when arguments to functions have no normal forms. For instance, consider the following term:

$$(\lambda x.(\lambda y.y))\ \Omega$$

If we use CBV semantics to evaluate the term, we must reduce Ω to a value before we can apply the function. But Ω never evaluates to a value, so we can never apply the function. Under CBV semantics, this term does not have a normal form.

If we use CBN semantics, then we can apply the function immediately, without needing to reduce the actual argument to a value. We have

$$(\lambda x.(\lambda y.y))\ \Omega \longrightarrow_{\text{CBN}} \lambda y.y$$

CBV and CBN are common evaluation orders; many programming languages use CBV semantics. So-called "lazy" languages, such as Haskell, typically use Call-by-need semantics, a more efficient semantics similar to CBN in that it does not evaluate actual arguments unless necessary. However, Call-by-need semantics ensures that arguments are evaluated at most once.

3 Lambda calculus encodings

The pure lambda calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure lambda calculus. We can however encode objects, such as booleans, and integers.

3.1 Booleans

We want to encode constants and operators for booleans. That is, we want to define functions *TRUE*, *FALSE*, *AND*, *IF*, and other operators such that the expected behavior holds, for example:

$$\begin{aligned} \mathit{AND}\ \mathit{TRUE}\ \mathit{FALSE} &= \mathit{FALSE} \\ \mathit{IF}\ \mathit{TRUE}\ e_1\ e_2 &= e_1 \\ \mathit{IF}\ \mathit{FALSE}\ e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining *TRUE* and *FALSE* as follows.

$$\begin{aligned} \mathit{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \mathit{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

Thus, both *TRUE* and *FALSE* take take two arguments, *TRUE* returns the first, and *FALSE* returns the second.

The function *IF* should behave like $\lambda b.\, \lambda t.\, \lambda f.\, \text{if } b = \mathit{TRUE} \text{ then } t \text{ else } f$. The definitions for *TRUE* and *FALSE* make this very easy.

$$\mathit{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

Definitions of other operators are also straightforward.

$$\begin{aligned} \mathit{NOT} &\triangleq \lambda b.\, b\ \mathit{FALSE}\ \mathit{TRUE} \\ \mathit{AND} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ b_2\ \mathit{FALSE} \\ \mathit{OR} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ \mathit{TRUE}\ b_2 \end{aligned}$$

3.2 Church numerals

Church numerals encode the natural number n as a function that takes f and x, and applies f to x n times.

$$\begin{aligned}
\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\
\overline{1} &= \lambda f.\, \lambda x.\, f\ x \\
\overline{2} &= \lambda f.\, \lambda x.\, f\ (f\ x) \\
\mathit{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)
\end{aligned}$$

In the definition for SUCC, the expression $n\ f\ x$ applies f to x n times (assuming that variable n is the Church encoding of the natural number n). We then apply f to the result, meaning that we apply f to x $n+1$ times.

Given the definition of SUCC, we can easily define addition. Intuitively, the natural number $n_1 + n_2$ is the result of apply the successor function n_1 times to n_2.

$$\mathit{ADD} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \mathit{SUCC}\ n_2$$

Similarly, we can define multiplication, by noting $n_1 \times n_2$ is the result of applying n_1 times to 0 that function that adds n_2 to its input. The latter can be obtained by considering $\mathit{ADD}\ n_2$, and thus

$$\mathit{MUL} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ (\mathit{ADD}\ n_2)\ \overline{0}$$

It is a lot more challenging to define subtraction. The difficulty is defining a function that takes a Church numeral representing n and returning its predecessor, the Church numeral representing $n-1$. It is possible to define such a function PRED. It is a difficult exercise, but the answer is easy enough to find online. Here's an intuition to get you started: think about enumerating the pairs $(0,1),(1,2),(2,3),(3,4),\ldots$. Finding the predecessor of n amounts to finding the nth pair in this enumeration, and looking at the first component of the pair.

How do we encode pairs, though? Funny you should ask...

3.3 Pairs

A pair (a,b) is a packaging up of two elements a and b in such a way that you can treat the package as a single unit, and retrieve both a and b later. This means that we're looking for a functions PAIR, FIRST, and SECOND with the property that:

$$\begin{aligned}
&\mathit{FIRST}(\mathit{PAIR}\ a\ b) = a \\
&\mathit{SECOND}(\mathit{PAIR}\ a\ b) = b
\end{aligned}$$

There are several encodings possible that satisfy this specification. The easiest is probably to encode a pair as a function that expects a “selector” and applies it to both elements of the pair. FIRST and SECOND then simply supply the appropriate selector to the pair.

$$\begin{aligned}
&\mathit{PAIR} = \lambda a.\, \lambda b.\, \lambda s.\, s\ a\ b \\
&\mathit{FIRST} = \lambda p.\, p\ (\lambda x.\, \lambda y.\, x) \\
&\mathit{SECOND} = \lambda p.\, p\ (\lambda x.\, \lambda y.\, y)
\end{aligned}$$

It is easy to generalize this encoding to arbitrary k-tuples, and even to arbitrary-sized lists with constructors NIL and CONS and accessors HEAD and TAIL.

4 Recursion and the fixed-point combinators

We can write nonterminating functions, as we saw with the expression Ω. We can also write recursive functions that terminate. However, one complication is how we express this recursion.

Let's consider how we would like to define a function that computes factorials.

$$FACT \triangleq \lambda n.\, IF\ (ISZERO\ n)\ 1\ (MUL\ n\ (FACT\ (PRED\ n)))$$

(We have not defined the predicate $ISZERO$. It is an easy exercise though.)

In slightly more readable notation (and we will see next lecture how we can translate more readable notation into appropriate expressions):

$$FACT \triangleq \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times FACT\ (n-1)$$

Here, like in the definitions we gave above, the name $FACT$ is simply meant to be shorthand for the expression on the right-hand side of the equation. But $FACT$ appears on the right-hand side of the equation as well! This is not a definition, it's a recursive equation.

4.1 Recursion removal trick

We can perform a "trick" to define a function $FACT$ that satisfies the recursive equation above. First, let's define a new function $FACT'$ that looks like $FACT$, but takes an additional argument f. We assume that the function f will be instantiated with an actual parameter of... $FACT'$.

$$FACT' \triangleq \lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ f\ (n-1))$$

Note that when we call f, we pass it a copy of itself, preserving the assumption that the actual argument for f will be $FACT'$.

Now we can define the factorial function $FACT$ in terms of $FACT'$.

$$FACT \triangleq FACT'\ FACT'$$

Let's try evaluating $FACT$ applied to an integer.

$$\begin{aligned}
FACT\ 3 &= (FACT'\ FACT')\ 3 && \text{Definition of } FACT \\
&= ((\lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ f\ (n-1)))\ FACT')\ 3 && \text{Definition of } FACT' \\
&\longrightarrow (\lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (FACT'\ FACT'\ (n-1)))\ 3 && \text{Application to } FACT' \\
&\longrightarrow \textsf{\textbf{if}}\ 3 = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ 3 \times (FACT'\ FACT'\ (3-1)) && \text{Application to } n \\
&\longrightarrow 3 \times (FACT'\ FACT'\ (3-1)) && \text{Evaluating } \textsf{\textbf{if}} \\
&\longrightarrow \ldots \\
&\longrightarrow 3 \times 2 \times 1 \times 1 \\
&\longrightarrow^* 6
\end{aligned}$$

So we now have a technique for writing a recursive function f: write a function f' that explicitly takes a copy of itself as an argument, and then define $f \triangleq f'\ f'$.

4.2 Fixed point combinators

There is another way of writing recursive functions: expressing the recursive function as the fixed point of some other, higher-order function, and then finding that fixed point.

Let's consider the factorial function again. The factorial function $FACT$ is a fixed point of the following function.

$$G \triangleq \lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ (n-1))$$

(Recall that if g if a fixed point of G, then we have $G\ g = g$.)

So if we had some way of finding a fixed point of G, we would have a way of defining the factorial function $FACT$.

There are such "fixed point operators," and the (infamous) Y combinator is one of them. Thus, we can define the factorial function $FACT$ to be simply $Y\ G$, the fixed point of G.

(A *combinator* is simply a closed lambda term; it is a higher-order function that uses only function application and other combinators to define a result from its arguments; our functions $SUCC$ and ADD are examples of combinators. It is possible to define programs using only combinators, thus avoiding the use of variables completely.)

The Y combinator is defined as

$$Y \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators.

The fixed point of the higher-order function G is equal to $G\ (G\ (G\ (G\ (G\ \ldots))))$. Intuitively, the Y combinator unrolls this equality, as needed. Let's see it in action, on our function G, where

$$G = \lambda f.\, \lambda n.\, \textsf{if } n = 0 \textsf{ then } 1 \textsf{ else } n \times (f\ (n-1))$$

and the factorial function is the fixed point of G. (We will use CBN semantics; see the note below.)

$$\begin{aligned}
FACT &= Y\ G \\
&= (\lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)))\ G && \text{Definition of } Y \\
&\longrightarrow (\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x)) \\
&\longrightarrow G\ (\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x))
\end{aligned}$$

Here, note that $(\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x))$ was the result of beta-reducing $Y\ G$. That is $(\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x))$ is β-equivalent to $Y\ G$ which is equal to $FACT$. So we will rewrite the expression as follows.

$$\begin{aligned}
&=_\beta G\ (FACT) \\
&= (\lambda f.\, \lambda n.\, \textsf{if } n = 0 \textsf{ then } 1 \textsf{ else } n \times (f\ (n-1)))\ FACT && \text{Definition of } G \\
&\longrightarrow \lambda n.\, \textsf{if } n = 0 \textsf{ then } 1 \textsf{ else } n \times (FACT\ (n-1))
\end{aligned}$$

Note that the Y combinator works under CBN semantics, but not CBV. What happens when we evaluate $Y\ G$ under CBV? Have a try and see. There is a variant of the Y combinator, Z, that works under CBV semantics. It is defined as

$$Z \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y)).$$

There are many (indeed infinite) fixed-point combinators. To gain some more intuition for fixed-point combinators, let's derive the Turing fixed-point combinator, discovered by Alan Turing, and denoted by Θ.

Suppose we have a higher-order function f, and want the fixed point of f. We know that $\Theta\ f$ is a fixed point of f, so we have

$$\Theta\ f = f\ (\Theta\ f).$$

This means, that we can write the following recursive equation for Θ.

$$\Theta = \lambda f.\, f\ (\Theta\ f)$$

.

Now we can use the recursion removal trick we described earlier! Let's define $\Theta' = \lambda t.\, \lambda f.\, f\ (t\ t\ f)$, and define

$$\begin{aligned}
\Theta &\triangleq \Theta'\ \Theta' \\
&= (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))
\end{aligned}$$

Let's try out the Turing combinator on our higher-order function G that we used to define $FACT$. Again, we will use CBN semantics.

$$\begin{aligned}
FACT &= \Theta\ G \\
&= ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f)))\ G \\
&\longrightarrow (\lambda f.\, f\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ f))\ G \\
&\longrightarrow G\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ G) \\
&= G\ (\Theta\ G) && \text{for brevity} \\
&= (\lambda f.\, \lambda n.\, \textsf{if}\ n = 0\ \textsf{then}\ 1\ \textsf{else}\ n \times (f\ (n-1)))\ (\Theta\ G) && \text{Definition of } G \\
&\longrightarrow \lambda n.\, \textsf{if}\ n = 0\ \textsf{then}\ 1\ \textsf{else}\ n \times ((\Theta\ G)\ (n-1)) \\
&= \lambda n.\, \textsf{if}\ n = 0\ \textsf{then}\ 1\ \textsf{else}\ n \times (FACT\ (n-1))
\end{aligned}$$

5 Definitional translation

We saw that the denotational semantics of IMP defined the meaning of IMP commands as mathematical functions from stores to stores. We can describe denotational semantics as a form of compilation, from IMP to mathematics. We now consider definitional translation, where we define the meaning of language constructs by translation to another language. This is a form of denotational semantics, but instead of the target language being mathematics, it is a simpler programming language. Note that definitional translation does not necessarily produce clean or efficient code; rather, it defines the meaning of the source language in terms of the target language.

We now consider a number of language features we can add to the lambda calculus, define an operational semantics for them, and then give an alternate semantics by translation to the simpler language that is the lambda calculus without additional features. We first introduce *evaluation contexts* to help us present the new language features succinctly.

5.1 Evaluation contexts

Recall the syntax and CBV operational semantics for the lambda calculus.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

Of the operational semantics rules, only the β-reduction rule told us how to "reduce" an expression; the other two rules were simply telling us the order to evaluate expressions in, i.e., first evaluate the left hand side of an application to a value, then evaluate the right hand side of an application to a value. The operational semantics of many of the languages we will consider have this feature: there are two kinds of rules, one kind specifying evaluation order, and the other kind specifying the "interesting" reductions.

Evaluation contexts provide us with a mechanism to separate out these two kinds of rules. An evaluation context E (sometimes written $E[\cdot]$) is an expression with a "hole" in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the "hole") in place of a subexpression. Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the language. The following grammar defines evaluation contexts for the pure CBV lambda calculus.

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

We write $E[e]$ to mean the evaluation context E where the hole has been replaced with the expression e. The following are examples of evaluation contexts, and evaluation contexts with the hole filled in by an expression.

$$\begin{aligned} E_1 &= [\cdot]\ (\lambda x.\, x) & E_1[\lambda y.\, y\ y] &= (\lambda y.\, y\ y)\ \lambda x.\, x \\ E_2 &= (\lambda z.\, z\ z)\ [\cdot] & E_2[\lambda x.\, \lambda y.\, x] &= (\lambda z.\, z\ z)\ (\lambda x.\, \lambda y.\, x) \\ E_3 &= ([\cdot]\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) & E_3[\lambda f.\, \lambda g.\, f\ g] &= ((\lambda f.\, \lambda g.\, f\ g)\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) \end{aligned}$$

Using evaluation contexts, we can define the evaluation semantics for the pure CBV lambda calculus with just two rules, one for evaluation contexts, and one for β-reduction.

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

Note that the evaluation contexts for the CBV lambda calculus ensure that we evaluate the left hand side of an application to a value, and then evaluate the right hand side of an application to a value before applying β-reduction.

We can specify the operational semantics of CBN lambda calculus using evaluation contexts:

$$E ::= [\cdot] \mid E\ e \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

We'll see the benefit of evaluation contexts as we see languages with more syntactic constructs.

5.2 Multi-argument functions and currying

Our syntax for functions restricted us to function that have a single argument: $\lambda x.\, e$. We could define a language that allows functions to have multiple arguments.

$$e ::= x \mid \lambda\langle x_1, \ldots, x_n\rangle.\, e \mid e_0\ \langle e_1, \ldots, e_n\rangle$$

Here, a function $\lambda\langle x_1, \ldots, x_n\rangle.\, e$ takes n arguments, with names x_1 through x_n. In a multi-argument application $e_0\ \langle e_1, \ldots, e_n\rangle$, we expect e_0 to evaluate to an n-argument function, and $e_1, \ldots, e_n$ are the arguments that we will give the function.

We can define a CBV operational semantics for the multi-argument lambda calculus as follows.

$$E ::= [\cdot] \mid E\ \langle e_1, \ldots, e_n\rangle \mid v_0\ \langle v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n\rangle$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda\langle x_1, \ldots, x_n\rangle.\, e_0)\ \langle v_1, \ldots, v_n\rangle \longrightarrow e_0\{v_1/x_1\}\{v_2/x_2\}\ldots\{v_n/x_n\}}$$

The evaluation contexts ensure that we evaluate a multi-argument application $e_0\ \langle e_1, \ldots, e_n\rangle$ by evaluating each expression from left to right down to a value.

Now, the multi-argument lambda calculus isn't any more expressive that the pure lambda calculus. We can show this by showing how any multi-argument lambda calculus program can be translated into an equivalent pure lambda calculus program. We define a translation function $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument lambda calculus and returns an equivalent expression in the pure lambda calculus. That is, if e is a multi-argument lambda calculus expression, $\mathcal{T}[\![e]\!]$ is a pure lambda calculus expression.

We define the translation as follows.

$$\begin{aligned} \mathcal{T}[\![x]\!] &= x \\ \mathcal{T}[\![\lambda\langle x_1, \ldots, x_n\rangle.\, e]\!] &= \lambda x_1.\, \ldots\ \lambda x_n.\, \mathcal{T}[\![e]\!] \\ \mathcal{T}[\![e_0\ \langle e_1, \ldots, e_n\rangle]\!] &= (\ldots((\mathcal{T}[\![e_0]\!]\ \mathcal{T}[\![e_1]\!])\ \mathcal{T}[\![e_2]\!])\ldots\ \mathcal{T}[\![e_n]\!]) \end{aligned}$$

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*. Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C. We could describe this function, using mathematical notation for domains of functions, as being an element of $A \times B \to C$. Currying this function produces a function that is an element of $A \to (B \to C)$. That is, the curried version of the function takes an argument from domain A, and returns a function that takes an argument from domain B and produces a result of domain C.

5.3 Products and let

We introduce two useful language features to the lambda calculus: products and let expressions.

A product is a pair of expressions (e_1, e_2). If e_1 and e_2 are both values, then we regard the product as also being a value. (That is, we cannot further evaluate a product if both elements are values.)

Given, a product, we can access the first or second element using the operators #1 and #2 respectively. That is, $\#1\ (v_1, v_2) \longrightarrow v_1$ and $\#2\ (v_1, v_2) \longrightarrow v_2$. (Other common notation for projection includes π_1 and π_2, and fst and snd.)

More formally, we define the syntax of lambda calculus with products and let expressions as follows. Values in this language are either functions or pairs of values.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ v &::= \lambda x.\, e \mid (v_1, v_2) \end{aligned}$$

We define a small-step CBV operational semantics for the language using evaluation contexts.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e_2$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\frac{}{\#1\ (v_1, v_2) \longrightarrow v_1} \qquad \frac{}{\#2\ (v_1, v_2) \longrightarrow v_2}$$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow e\{v/x\}}$$

We can give an equivalent semantics by translation to the pure CBV lambda calculus. We encode a pair (e_1, e_2) using the same encoding we saw earlier in lecture, as a value that takes a function f, and applies f to v_1 and v_2, where v_1 and v_2 are the result of evaluating e_1 and e_2 respectively. The projection operators pass a function to the encoding of pairs that selects either the first or second element as appropriate.

Note also that the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is equivalent to the application $(\lambda x.\, e_2)\ e_1$.

$$\begin{aligned} \mathcal{T}[\![x]\!] &= x \\ \mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\ \mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\ \mathcal{T}[\![(e_1, e_2)]\!] &= (\lambda x.\, \lambda y.\, \lambda f.\, f\ x\ y)\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\ \mathcal{T}[\![\#1\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, x) \\ \mathcal{T}[\![\#2\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, y) \\ \mathcal{T}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= (\lambda x.\, \mathcal{T}[\![e_2]\!])\ \mathcal{T}[\![e_1]\!] \end{aligned}$$

5.4 CBN to CBV

We've seen semantics for both the call-by-name lambda calculus and the call-by-value lambda calculus. We can translate a call-by-name program into a call-by-value program. In CBV, arguments to functions are evaluated before the function is applied; in CBN, functions are applied as soon as possible. In the translation, we delay the evaluation of arguments by wrapping them in a function. This is called a *thunk*: wrapping a computation in a function to delay its evaluation.

Since arguments to functions are turned into thunks, when we want to use an argument in a function body, we need to evaluate the thunk. We do so by applying the thunk (which is simply a function); it doesn't matter what we apply the thunk to, since the thunk's argument is never used.

$$\begin{aligned}
\mathcal{T}[\![x]\!] &= x\ (\lambda y.\, y) \\
\mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ (\lambda z.\, \mathcal{T}[\![e_2]\!]) && z \text{ is not a free variable of } e_2
\end{aligned}$$

5.5 Adequacy of translation

We've presented several translations of languages. In each case, we had a semantics defined for both the source and target language. We would like the translation to be correct, that is, to preserve the meaning of source programs.

More precisely, we would like an expression e in the source language to evaluate to a value v if and only if the translation of e evaluates to a value v' such that v' is "equal to" v.

What exactly it means for v' to be "equal to" v will depend on the translation. Sometimes, it will mean that v' is the translation of v; other times, it will mean that v' is somehow equivalent to the translation of v. In particular, we often need to define equivalence on functions. One possible solution is that two functions are equivalent if they agree on the result when applied to any value of a base type (e.g., integers or booleans). The idea is that if two functions disagree when passed a more complex value (say, a function), then we could write a program that uses these functions to produce functions that disagree on values of base types.

There are two criteria for a translation to be *adequate*: soundness and completeness. For clarity, let's suppose that $\mathbf{Exp}_{\text{src}}$ is the set of source language expressions, and that $\longrightarrow_{\text{src}}$ and $\longrightarrow_{\text{trg}}$ are the evaluation relations for the source and target languages respectively.

A translation is sound if every target evaluation represents a source evaluation:

$$\text{Soundness: } \forall e \in \mathbf{Exp}_{\text{src}}.\ \text{if } \mathcal{T}[\![e]\!] \longrightarrow^*_{\text{trg}} v' \text{ then } \exists v.\ e \longrightarrow^*_{\text{src}} v \text{ and } v' \text{ equivalent to } v$$

A translation is complete if every source evaluation has a target evaluation.

$$\text{Completeness: } \forall e \in \mathbf{Exp}_{\text{src}}.\ \text{if } e \longrightarrow^*_{\text{src}} v \text{ then } \exists v'.\ \mathcal{T}[\![e]\!] \longrightarrow^*_{\text{trg}} v' \text{ and } v' \text{ equivalent to } v$$