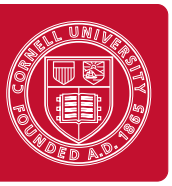
CS 4110 – Programming Languages and Logics
Lectures #12-14: Programming in λ-calculus



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 Nontermination

Consider the expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, which we will refer to as omega for brevity. Let's try evaluating omega.

$$\begin{array}{rcl} \mathsf{omega} & = & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \rightarrow & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & = & \mathsf{omega} \end{array}$$

Evaluating omega never reaches a value! It is an infinite loop!

What happens if we use omega as an actual argument to a function? Consider the following program.

$$(\lambda x.(\lambda y.y))\ \mathsf{omega}$$

If we use CBV semantics to evaluate the program, we must reduce omega to a value before we can apply the function. But omega never evaluates to a value, so we can never apply the function. Thus, under CBV semantics, this program does not terminate. If we use CBN semantics, we can apply the function immediately, without needing to reduce the actual argument to a value:

$$(\lambda x.(\lambda y.y))\ \mathsf{omega} \rightarrow_{\text{CBN}} \lambda y.y$$

CBV and CBN are common evaluation orders; many functional programming languages use CBV semantics. Later we will see the call-by-need strategy, which is similar to CBN in that it does not evaluate actual arguments unless necessary but is more efficient.

2 λ-calculus encodings

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

2.1 Booleans

Let us start by encoding constants and operators for booleans. That is, we want to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as expected. For example:

$$\begin{aligned} \mathsf{AND\ TRUE\ FALSE} &= \mathsf{FALSE} \\ \mathsf{NOT\ FALSE} &= \mathsf{TRUE} \\ \mathsf{IF\ TRUE}\ e_1\ e_2 &= e_1 \\ \mathsf{IF\ FALSE}\ e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\mathsf{TRUE} \triangleq \lambda x.\, \lambda y.\, x$$
$$\mathsf{FALSE} \triangleq \lambda x.\, \lambda y.\, y$$

Thus, both TRUE and FALSE are functions that take two arguments; TRUE returns the first, and FALSE returns the second. We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\, \text{if } b = \mathsf{TRUE} \text{ then } t \text{ else } f.$$

The definitions for TRUE and FALSE make this very easy.

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

Definitions of other operators are also straightforward.

$$\mathsf{NOT} \triangleq \lambda b.\, b\ \mathsf{FALSE}\ \mathsf{TRUE}$$
$$\mathsf{AND} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\ b_2\ \mathsf{FALSE}$$
$$\mathsf{OR} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\ \mathsf{TRUE}\ b_2$$

2.2 Church numerals

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

$$\overline{0} \triangleq \lambda f.\, \lambda x.\, x$$
$$\overline{1} = \lambda f.\, \lambda x.\, f\ x$$
$$\overline{2} = \lambda f.\, \lambda x.\, f\ (f\ x)$$
$$\mathsf{SUCC} \triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$$

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.

$$\mathsf{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \mathsf{SUCC}\ n_2$$

3 Recursion

We can write nonterminating functions, as we saw with omega. We can also write recursive functions that terminate. However, we need to develop techniques for expressing recursion.

Let's consider how we would like to write the factorial function.

$$\mathsf{FACT} \triangleq \lambda n.\, \mathsf{IF}\ (\mathsf{ISZERO}\ n)\ 1\ (\mathsf{TIMES}\ n\ (\mathsf{FACT}\ (\mathsf{PRED}\ n)))$$

In slightly more readable notation, this is just:

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

Here, as in the definition 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.

3.1 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 FACT' itself.

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

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'.

$$\mathsf{FACT} \triangleq \mathsf{FACT}'\ \mathsf{FACT}'$$

Let's try evaluating FACT on an integer.

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

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 $f \triangleq f'\ f'$.

3.2 Fixed point combinators

There is another way of writing recursive functions: we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point. We saw this technique earlier in the course when we defined the denotational semantics for **while** loops.

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

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

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 a number of "fixed point combinators," and the (infamous) Y combinator is one of them. Thus, we can define the factorial function FACT to be simply $\mathsf{Y}\ G$, the fixed point of G. The Y combinator is defined as

$$\mathsf{Y} \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators. Note how similar its defnition is to omega.

We'll use a slight variant of the Y combinator, Z, which is easier to use under CBV. (What happens when we evaluate $\mathsf{Y}\ G$ under CBV?). The Z combinator is defined as

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

Let's see it in action, on our function G. Define FACT to be Z G and calculate as follows:

$$
\begin{array}{lll}
 & \mathsf{FACT} & \\
= & \mathsf{Z}\ G & \\
= & (\lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y)))\ G & \text{Definition of Z} \\
\to & (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y)) & \\
\to & G\ (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) & \\
= & (\lambda f.\, \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\ (n-1))) & \\
 & \quad (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) & \text{definition of } G \\
\to & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1 & \\
 & \quad \textbf{else}\ n \times ((\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y)\ (n-1)) & \\
=_\beta & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\lambda y.\, (\mathsf{Z}\ G)\ y)\ (n-1) & \\
=_\beta & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathsf{Z}\ G\ (n-1)) & \\
= & \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathsf{FACT}\ (n-1)) &
\end{array}
$$

There are many (indeed infinitely many) fixed-point combinators. Here's a cute one:

$$\mathsf{Y}_k \triangleq (\mathsf{L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L})$$

where

$$\mathsf{L} \triangleq \lambda abcdefghijklmnopqstuvwxyzr.\, (r\ (t\ h\ i\ s\ i\ s\ a\ f\ i\ x\ e\ d\ p\ o\ i\ n\ t\ c\ o\ m\ b\ i\ n\ a\ t\ o\ r))$$

To gain some more intuition for fixed-point combinators, let's derive a fixed-point combinator that was originally discovered by Alan Turing. 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:

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

Now we can use the recursion removal trick we described earlier. Define Θ' as $\lambda t.\, \lambda f.\, f\ (t\ t\ f)$, and Θ as $\Theta'\ \Theta'$. Then we have the following equalities:

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

Let's try out the Turing combinator on our higher order function G that we used to define FACT.

This time we will use CBN evaluation.

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

4 Definitional translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus. 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 being mathematical objects, it is a simpler programming language (such as λ-calculus). Note that definitional translation does not necessarily produce clean or efficient code; rather, it defines the meaning of the source language constructs in terms of the target language.

For each language construct, we will define an operational semantics directly, and then give an alternate semantics by translation to a simpler language. We will start by introducing *evaluation contexts*, which make it easier to present the new language features succinctly.

4.1 Evaluation contexts

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

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \\
v &::= \lambda x.\, e
\end{aligned}
$$

$$
\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}
$$

Of the operational semantics rules, only the β-reduction rule told us how to "reduce" an expression; the other two rules tell us the order to evaluate expressions—e.g., evaluate the left hand side of an application to a value first, 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, *congruence rules* that specify evaluation order, and the *computation rules* that specify the "interesting" reductions.

Evaluation contexts are a simple mechanism that separates 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 λ-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.

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

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

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

Note that the evaluation contexts for the CBV λ-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 λ-calculus using evaluation contexts:

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

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

4.2 Multi-argument functions and currying

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

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

Here, a function $\lambda x_1, \ldots, x_n.\, e$ takes n arguments, with names x_1 through x_n. In a multi argument application $e_0\ e_1\ \ldots\ e_n$, 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 λ-calculus as follows.

$$E ::= [\cdot] \mid v_0\ \ldots\ v_{i-1}\ E\ e_{i+1}\ \ldots\ e_n \qquad \frac{e \to e'}{E[e] \to E[e']}$$

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda x_1, \ldots, x_n.\, e_0)\ v_1\ \ldots\ v_n \to e_0\{v_1/x_1\}\{v_2/x_2\}\ldots\{v_n/x_n\}}$$

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

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

We define the translation as follows.

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

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.

4.3 Products and let

We introduce two useful language features to the λ-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, $\#1\ (v_1, v_2) \to v_1$ and $\#2\ (v_1, v_2) \to v_2$. (Other common notation for projection includes π_1 and π_2, and fst and snd.)

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

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

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

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

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

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

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

We can give an equivalent semantics by translation to the pure CBV λ-calculus. Note that we encode a pair (e_1, e_2) 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 $(\lambda x.\, e_2)\ e_1$.

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

4.4 Encoding call-by-name in call-by-value

We've seen semantics for both the call-by-name λ-calculus and the call-by-value λ-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.

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

4.5 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, there are many ways to define equivalences on functions. One way is to say 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 $\rightarrow_{\text{src}}$ and $\rightarrow_{\text{trg}}$ are the evaluation relations for the source and target languages respectively.

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

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

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

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

5 References

We introduce constructs for creating, reading, and updating memory locations, also called *references*. The resulting language is still a functional language (since functions are first-class values), but expressions can have side-effects, that is, they can modify state.

The syntax is defined as follows.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \ell \\ v &::= \lambda x.\, e \mid \ell \end{aligned}$$

Expression $\mathsf{ref}\ e$ creates a new memory location (like a malloc), and sets the initial contents of the location to (the result of) e. The expression $\mathsf{ref}\ e$ itself evaluates to a memory location ℓ. Think of a location as being like a pointer to a memory address. The expression $!e$ assumes that e evaluates to a memory location, and $!e$ evaluates to the current contents of the memory location. Expression $e_1 := e_2$ assumes that e_1 evaluates to a memory location ℓ, and updates the contents of ℓ with (the result of) e_2. Locations ℓ are not intended to be used directly by a programmer: they are not part of the *surface syntax* of the language, the syntax that a programmer would write. They are introduced only by the operational semantics.

We define a small-step CBV operational semantics. We use configurations $\langle e, \sigma \rangle$, where e is an expression, and σ is a map from locations to values.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \mathsf{ref}\ e \mid !E \mid E := e \mid v := E \qquad \frac{\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle}{\langle E[e], \sigma \rangle \rightarrow \langle E[e'], \sigma' \rangle}$$

$$\beta\text{-REDUCTION}\ \frac{}{\langle (\lambda x.\, e)\ v, \sigma \rangle \rightarrow \langle e\{v/x\}, \sigma \rangle} \qquad \text{ALLOC}\ \frac{}{\langle \mathsf{ref}\ v, \sigma \rangle \rightarrow \langle \ell, \sigma[\ell \mapsto v] \rangle}\ \ell \notin dom(\sigma)$$

$$\text{DEREF}\ \frac{}{\langle\ !\ell, \sigma \rangle \rightarrow \langle v, \sigma \rangle}\ \sigma(\ell) = v \qquad \text{ASSIGN}\ \frac{}{\langle \ell := v, \sigma \rangle \rightarrow \langle v, \sigma[\ell \mapsto v] \rangle}$$

References do not add any expressive power to the λ-calculus: it is possible to translate λ-calculus with references to the pure λ-calculus. Intuitively, this is achieved by explicitly representing the store, and threading the store through the evaluation of the program. The details are left as an exercise.

6 Continuations

So far we have seen a number of language features that extend lambda calculus, and have translated many of these into the pure lambda calculus, using a direct semantic translation. That is, the control structure of the source language translated into the corresponding control structure in the target language:

$$\mathcal{T}[\![\lambda x.\, e]\!] = \lambda x.\, \mathcal{T}[\![e]\!]$$
$$\mathcal{T}[\![e_1\ e_2]\!] = \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!]$$

This style of translation works well when the source language is similar to the target language. However, when the control structures of the source and target languages differ considerable, it doesn't work as well.

In this lecture, we'll learn about *continuations* and *continuation-passing style*, and then, in later lectures, use these notions to define the semantics of control-flow constructs such as exceptions.

Continuations are a programming technique that may be used directly by a programmer, or used in program transformations by a compiler.

Intuitively, a continuation represents "the rest of the program." Consider the program

$$\mathsf{if}\ \mathsf{foo} < 10\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar}$$

and consider the evaluation of the expression $\mathsf{foo} < 10$. When we finish evaluating this subexpression, we will evaluate the if statement, and then evaluate the appropriate branch. The *continuation* of the subexpression $\mathsf{foo} < 10$ is the rest of the computation that will occur after we evaluate the subexpression. We can write this continuation as a function that takes the result of the subexpression:

$$(\lambda y.\, \mathsf{if}\ y\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar})\ (\mathsf{foo} < 10)$$

The evaluation order and result of this program will be the same as the original expression; the difference is that we extracted the continuation of the subexpression in to a function.

The nice thing about continuations is that it makes the control explicit, and this is especially useful in the case of functional programs, where control is not explicit otherwise. In fact, we can rewrite a program to make continuations more explicit.

Let's consider another program, and convert it so that continuations are explicit. Let's think about the following applied λ-calculus program.

$$(\lambda x.\, x)\ ((1 + 2) + 3) + 4$$

We'll start by defining a continuation for the outermost evaluation context, which takes a value, and applies the identity function to it.

$$k_0 = \lambda v.\, (\lambda x.\, x)\ v$$

The evaluation context that is evaluated next-to-last takes a value, adds 4 to it, and then passes the result to k_0.

$$k_1 = \lambda a.\, k_0\ (a + 4)$$

Likewise, for the next evaluation contexts.

$$k_2 = \lambda b.\, k_1\ (b + 3)$$
$$k_3 = \lambda c.\, k_2\ (c + 2)$$

The program itself is now equivalent to $k_3\ 1$. Since $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is just syntactic sugar for $(\lambda x.\, e')\ e$, we can actually rewrite the above as

$$\begin{array}{l}\mathsf{let}\ c = 1\ \mathsf{in}\\ \mathsf{let}\ b = c + 2\ \mathsf{in}\\ \mathsf{let}\ a = b + 3\ \mathsf{in}\\ \mathsf{let}\ v = a + 4\ \mathsf{in}\\ (\lambda x.\, x)\ v\end{array}$$

This is fairly close to some machine instructions of the form:

$$\begin{array}{l}\mathsf{set}\ c, 1\\ \mathsf{add}\ b, c, 2\\ \mathsf{add}\ a, b, 3\\ \mathsf{add}\ v, a, 4\\ \mathsf{call\ id}, v\end{array}$$

Using continuations, functions can be transformed into "functions that don't return"—i.e., functions that take, besides the usual arguments, an additional argument representing a continuation. When the function finishes, it invokes the continuation on its result, instead of returning the result to its caller. Writing functions in this way is usually referred to as Continuation-Passing Style, or CPS for short. For instance, the CPS version of factorial looks like the following:

$$\mathsf{FACT}_{cps} = \mathsf{Y}\ \lambda f.\, \lambda n, k.\, \mathsf{if}\ n = 0\ \mathsf{then}\ k\ 1\ \mathsf{else}\ f\ (n - 1)\ (\lambda v.\, k\ (n * v))$$

Note that the last thing that code in FACT_{cps} does is call a function (either k or f), and does not do anything with the result.

Continuation-Passing Style is an important concept in the compilation of functional languages and is used as an intermediate compiler representation (it has been used in compilers for Scheme, ML, etc). The main advantage is that CPS makes the control flow explicit and makes it easier to translate functional code to machine code where control is explicit (in the form of sequences of machine instructions and jumps). For instance, a CPS call can be easily translated into a jump to the invoked method, since the invoked function does not return the control.

6.1 CPS translation

We can translate λ-calculus programs into continuation-passing style. We define a translation function $\mathcal{CPS}[\![\cdot]\!]$, which takes a CBV λ-calculus expression, and translates the expression to a CBV λ-calculus expression in continuation-passing style.

Let's consider a translation from λ-calculus with pairs and integers. The syntax of the source language is as follows.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

The translation $\mathcal{CPS}[\![e]\!]$ will produce a function that whose argument is the continuation to which to pass the result. That is, for all expressions e, the translation is of the form $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$, where k is a continuation. We will both assume and guarantee that for any expression e, the translation $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$ will apply k to the result of evaluating e.

For convenience, instead of writing

$$\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$$

we write

$$\mathcal{CPS}[\![e]\!]\ k = \ldots$$

$$\begin{aligned}
\mathcal{CPS}[\![n]\!]\ k &= k\ n \\
\mathcal{CPS}[\![e_1 + e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda n.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda m.\, k\ (n + m))) && n \text{ is not a free variable of } e_2 \\
\\
\mathcal{CPS}[\![(e_1, e_2)]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda v.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda w.\, k\ (v, w))) && v \text{ is not a free variable of } e_2 \\
\mathcal{CPS}[\![\#1\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#1\ v)) \\
\mathcal{CPS}[\![\#2\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#2\ v)) \\
\\
\mathcal{CPS}[\![x]\!]\ k &= k\ x \\
\mathcal{CPS}[\![\lambda x.\, e]\!]\ k &= k\ (\lambda x, k'.\, \mathcal{CPS}[\![e]\!]k') && k' \text{ is not a free variable of } e \\
\mathcal{CPS}[\![e_1\ e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda f.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda v.\, f\ v\ k)) && f \text{ is not a free variable of } e_2
\end{aligned}$$

We translate a function $\lambda x.\, e$ to a function that takes an additional argument k', which is the continuation after the function application. That is, k' is the continuation to which we hand the result of evaluating the function body. In function application, we see that in addition to the actual argument, we also give the continuation as the additional argument.

Let's see an example translation and execution...

$$
\begin{aligned}
\mathcal{CPS}[\![(\lambda a.\, a + 6)\ 7]\!]\ \mathsf{ID} &= \mathcal{CPS}[\![(\lambda a.\, a + 6)]\!]\ \ (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ v\ \mathsf{ID})) \\
&= (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ v\ \mathsf{ID}))\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!]k') \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!]k') \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!]\ (\lambda n.\, \mathcal{CPS}[\![6]\!]\ (\lambda m.\, k'\ (m + n)))) \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!]\ (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)) \\
&= (\lambda f.\, (\lambda v.\, f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a) \\
&\to (\lambda v.\, (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a)\ v\ \mathsf{ID})\ 7 \\
&\to (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a)\ 7\ \mathsf{ID} \\
&\to (\lambda n.\, (\lambda m.\, \mathsf{ID}\ (m + n))\ 6)\ 7 \\
&\to (\lambda m.\, \mathsf{ID}\ (m + 7))\ 6 \\
&\to \mathsf{ID}\ (6 + 7) \\
&\to \mathsf{ID}\ 13 \\
&\to 13
\end{aligned}
$$