CS 4110 – Programming Languages and Logics
Lecture #16: Programming in the λ-calculus

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:

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

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 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.

2.1 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.

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

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 $f \triangleq f'\ f'$.

2.2 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 $\mathsf{Z}\ G$ and calculate as follows:

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

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:

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

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.

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

3 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.

3.1 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.

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

Using evaluation contexts, we can define the evaluation semantics for the pure CBV 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.

3.2 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.

3.3 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.

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

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

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

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

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 $(\lambda x.\, e_2)\ e_1$.

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