Combinators and Recursion

CS 440: Programming Languages and Translators, Fall 2020

A. Combinators

• In the lambda calculus, a combinator is a function that contains no free (i.e., non-local variables). E.g., \( \lambda x . x \) and \( \lambda x y . x \) are both combinators but \( \lambda x . x y \) is not because the occurrence of \( y \) is free (not bound to a lambda). Since combinators don’t introduce variables, it’s easy to combine them. **Combinatory logic** is the study of combinators.

• Syntax: A combinator an expression built up from variables, basic combinators, and function application. (The variables have combinators as values.)
  - \( \text{Combinator} \to \text{Variable} \mid \text{Basic} \mid \text{Application} \)
  - \( \text{Application} \to (\text{Combinator} \ \text{Combinator}) \) (left associative)
  - \( \text{Basic} \to S \mid K \mid I \)

• There are various different collections of basic combinators, but \( S, K, I \) are the usual ones because it turns out that every combinator can be built using these three.

• **Semantics of \( S, K, I \)**
  - \( I x = x \)
  - \( K x y = x \)
  - \( S x y z = (x z) y z \)

• You can write \( \lambda \) expressions to describe combinators; in that case, \( I = \lambda x . x \), \( K = \lambda x y . x \), and \( S = \lambda x y z . x z (y z) \).

• **Notation**: Let’s write \( I x \to x \) ("\( I x \) reduces to \( x \)") if we want to emphasize that a combinator calculation is going on. An arrow other direction reverses things \( x \leftarrow I x \) ("\( x \) expands to \( I x \)") is the same as \( I x \to x \).

• **Notation**: When people write \( expr_1 = expr_2 \), they generally mean (1) Either \( expr_1 \to expr_2 \) or \( expr_1 \leftarrow expr_2 \) (we get to work out which) or (2) \( expr_1 \) and \( expr_2 \) are the endpoints of a sequence of expressions connected by \( \to \) and/or \( \leftarrow \) arrows with.

• **Example 1**: \( K x y = x \) because \( K x y \to x \), and \( x = K x x \) because \( x \leftarrow K x x \), \( K x y = K x x \). (And we can’t replace the \( = \) by \( \to \) or \( \leftarrow \).)

• **Example 2**: The \( I \) combinator can actually be built using \( S \) and \( K \), so \( I \) isn’t strictly necessary.
  - \( S K K x = K x (K x) = x \leftarrow I x \), so \( I = S K K \).
  - In fact, we can use any combinator \( y \) instead of the second \( K \): \( S K y x = K x (y x) = x = I x \), so \( I = S K y \).

• **Example 3**: \( K = S K S K \). Proof: \( S K S K = K K (S K) = K \).
B. Normal Forms

- A combinator expression is in normal form if we can't apply any of the reduction rules for S, K, and I. (E.g., K I I is in normal form but K I I → I.)

- **Example 4:** First, S I I x = x x for all x. (Proof: S I I x = I x (I x) = x x.) If we use S I I as x, then S I I (S I I) = S I I (S I I) = S I I (S I I) = .... Since we can reduce S I I (S I I) any number of times, it doesn't have a normal form.

- **Notation:** Writing \( \text{expr}_1 \rightarrow^n \text{expr}_2 \) means that \( \text{expr}_1 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n \rightarrow \text{expr}_2 \) for some expressions \( e_1, e_2, \ldots \). Writing \( \text{expr}_1 \rightarrow^* \text{expr}_2 \) means that for some \( n, \text{expr}_1 \rightarrow^n \text{expr}_2 \). As an example, we can write \( \text{S I I (S I I)} \rightarrow^3 \text{S I I (S I I)} \rightarrow^3 \text{S I I (S I I)} \rightarrow^3 \ldots \) to emphasize that \( \text{S I I (S I I)} \) has no normal form because it reduces to itself over and over.

C. Recursive Functions, Fixed-Point Combinators, and the Y Combinator

- A family of factorial-like functions: Let \( \text{ff} = \lambda g \ n . \ (\text{if } n = 0 \ \text{then } 1 \ \text{else } n \ast g(n-1)) \), so \( \text{ff} \) has the framework of a factorial function except that it uses \( g \) to make the "recursive" call: E.g., \( f (\lambda m . m^2) = \lambda n . \ (\text{if } n = 0 \ \text{then } 1 \ \text{else } (n-1)^2) \), which of course, isn't the factorial function. Running \( \text{ff} \) on a family of different \( g \) gives a bunch of different factorial-like functions as results.

- If we do have a factorial function \( \text{fact} \) lying around, then \( \text{ff} \ \text{fact} = \text{fact} \) because \( \text{ff} \ \text{fact} \) is the skeleton of the factorial function with \( \text{fact} \) used to make recursive calls.

- **Definition:** A fixed point of a function \( f \) is a value \( p \) such that \( f \ p = p \).

- **Definition:** A fixed-point combinator \( \text{fix} \) is a combinator where (for all \( f \)), \( \text{fix} \ f = \text{fix} (\lambda g . \ g \ n . \ (\text{if } n = 0 \ \text{then } 1 \ \text{else } n \ast g(n-1))) \); \( i.e., \ (\text{fix} \ f) \) is a fixed point of \( f \).

- Going back to our example, \( \text{ff} \ g \) is a function that has the framework of factorial but uses \( g \) to make recursive calls. Then the factorial function \( \text{fact} \) has the property that \( \text{ff} \ \text{fact} = \text{fact} \). I.e., the factorial function is the fixed point of \( \text{ff} \) because it's the framework of factorial plus it uses \( \text{fact} \) to make recursive calls.

- **Non-recursive let:** You'll recall that the nonrecursive \( (\text{let } x = e_1 \ \text{in } e_2) = (\lambda x . e_2) (e_1) \), and \( (\text{let } f x = e_1 \ \text{in } e_2) = (\text{let } f = \lambda x . e_1 \ \text{in } e_2) = (\lambda f . e_2) (\lambda x . e_1) \). If expression \( e_2 \) uses "f", it gets the bound \( \lambda f \), as it should. But \( (\lambda x . e_1) \) is not in the scope of \( \lambda f \), so if \( e_1 \) uses \( f \), it's not the same \( f \). I.e., \( f \) is not defined recursively in \( e_1 \).

- **Recursive let:** To implement \( (\text{let rec } f = \lambda x . e_1 \ \text{in } e_2) \), we need to ensure that any \( f \) that appear in \( e_1 \) are bound to a function that behaves like a recursive \( f \). (It doesn't have to literally be \( f \), it's sufficient to behave like a recursive \( f \).) Since we just looked at fixed points, you're probably thinking that we need one here, and you're right. We have \( \text{fact} = \text{fix} (\lambda g n . \ldots g(n-1)) \), so \( (\text{let rec } \ \text{fact} = \lambda x . e_1 \ \text{in } e_2) \) needs \( \text{fact} = \text{fix} (\lambda \text{fact} . \lambda x . e_1) \). More generally, \( (\text{let rec } f = \lambda x . e_1 \ \text{in } e_2) \) needs \( f = \text{fix} (\lambda f . \lambda x . e_1) \) when it executes \( e_2 \). So our translation is \( (\lambda f . e_2) (\text{fix} \lambda f . \lambda x . e_1) \).

- **Definition:** \( \text{let rec } v = e_1 \ \text{in } e_2 \) means \( (\lambda v . e_2) (\text{fix} \lambda v . e_1) \). [This definition allows \( v \) to be recursively defined but not be a function; a recursive list, for example.]
The Y combinator: There are any number of fixed-point combinators, but probably the most famous is the Y combinator: $Y = \lambda f. (\lambda x. f(x \ x))(\lambda x. f(x \ x))$. It's reasonably straightforward to prove that Y is a fixed-point combinator (i.e., that for all f, $Y f = f(Y f)$). [It's not $Y f \rightarrow^* f(Y f)$, however; you need a combination of $\rightarrow$ and $\leftarrow$.]

References

- We're looking only at the very basics of combinators, so the wikipedia articles below give more detail than we need, but they're still quite useful.
  - https://en.wikipedia.org/wiki/Let_expression
  - https://en.wikipedia.org/wiki/Combinatory_logic