1 The Lambda Calculus (Continued)

1.1 Semantics: What Programs Mean

In particular, how do you run or execute a λ-term?

Unlike programming languages like Java or Python, where code runs sequentially, one can *think* of running a Lambda Calculus program as middle-school algebra.

```
(x + 2) * (3x - 1)
    => 3x^2 - x + 6x - 2    -- multiply polynomials
    => 3x^2 + 5x - 2        -- add monomials
                            -- no more rules apply
```

Essentially, rewrite step-by-step following simple rules until no more rules apply.

1.2 Scope of Variables

The scope of a variable is simply where, in a program, the variable is visible. In the expression `\x -> E`:

- `x` is the newly introduced variable.
- `E` is the **scope** of `x`.
- any occurrence of `x` in `\x -> E` is **bound** (by the **binder** `\x`).

For example, `x` is bound in:

```
\x -> x
\x -> (\y -> x)     -- The 'x' in the abstraction is bound by \x
                    -- Same thing as: \x y -> x
```

We say that an occurrence of `x` in `E` is **free** if it's not bound by an enclosing abstraction. For example:

```
x y                     -- No binders at all
\y -> x y               -- No \x binder
(\x -> \y -> y) x       -- x is outside of the scope of the \x binder
                        -- It's not "the same" 'x'
                        -- The outer x is not in the abstraction
```

Consider the expression `(\x -> x) x`. Here, there are two occurrences[1] of `x`; The `\x` is a parameter (i.e. declaration) of which the second `x` is the first occurrence. The third `x` is another occurrence. So, the very last `x` is a free occurrence and the second `x` (the one after the arrow) is bound to `\x`.

1.2.1 Free Variables

A variable `x` is **free** in `E` if there exists a free occurrence of `x` in `E`. We can formally define the set of all free variables in a term using this function:

$$FV(\texttt{Exp}) = \begin{cases} \{\texttt{x}\} & \text{If } \texttt{Exp = x} \\ FV(\texttt{E}) \setminus \{\texttt{x}\} & \text{If } \texttt{Exp = \x -> E} \\ FV(\texttt{E1}) \cup FV(\texttt{E2}) & \text{If } \texttt{Exp = E1 E2} \end{cases}$$

Here, `Exp` is some arbitrary expression.

[1] A use of the variable.

1.2.2 Closed Expressions

If `E` has no free variables, it is said to be **closed**. Closed expressions are also called **combinators**.

The shortest closed expression is just

```
\x -> x
```

since `x` is bound to by `\x`.

1.3 The Rewrite Rules of Lambda Calculus

1.3.1 Semantics: Beta Step

β-step is known as calling a function, or **function calls**. Formally, we write this as

```
(\x -> E1) E2 =b> E1[x := E2]
```

where `E1[x := E2]` means that `E1` with all *free* occurrences of `x` is replaced with `E2`. Effectively, the computation is done by search-and-replace:

- If you see an abstraction applied to an argument, take the body of the abstraction and replace all free occurrences of the formal by that argument.

Thus, the above formal definition is saying that `(\x -> E1) E2` β-steps to `E1[x := E2]`.

A few other examples are:

- Consider `(\f -> f (\x -> x)) (give apple)`. The idea is:
  - We take the body of the abstraction, i.e. `f (\x -> x)`.
  - We replace all instances of `f` (the formal parameter) in the body with `(give apple)`.
  - This gives us `(give apple) (\x -> x)`.
- Consider `(\x -> (\y -> y)) apple`. The idea is:
  - We take the body of the abstraction, i.e. `(\y -> y)`.
  - We replace all instances of `x`, the formal parameter, in the body with `apple`.
  - This gives us `\y -> y` (nothing has changed).

  5
- We take the body of the abstraction, i.e. `x (\x -> x)`. Note that there is another abstraction with the same parameter name as the original formal parameter name.
- `(\x -> x) y` becomes `y`.
- `(\a b c -> b) d` becomes `(\b c -> b)`.
- `(\b c -> b) e` becomes `(\c -> e)`.
- `(\a b c -> b) d e` becomes `(\c -> e)`.
- `(\x y z -> z y x) y z x` cannot be reduced; instead, we need to use what's known as an α-step.

1.3.2 Alpha Step

α-step is known as **renaming formals**. That is, we can rename a formal parameter and replace all its occurrences in the body.

For example:

```
(\x y z -> z y x) y z x
=a> (\x a z -> z a x) y z x
```

Formally, we write this as:

```
\x -> E     =a>     \y -> E[x := y]          where not (y in FV(E))
```

In particular, we note that the following expressions are α-equivalent:

```
\x -> x     =a>     \y -> y     =a>     \z -> z
```

Consider the following invalid examples.

- `\f -> fx =a> \x -> x x`: Changing `f` to `x` introduces a conflict with the other `x`.
- `(\x -> \y -> y) y =a> (\x -> \z -> \z) z`: When changing `y` to `z`, we also changed the `y` that's outside the scope of the inner body function; this is not allowed.
- `\x -> \y -> x y =a> \apple -> \orange -> apple orange`: There's no issue here, although it might be ideal to do one *alpha*-reduction at a time instead of two at a time.