The Lambda Calculus

The lambda-calculus

- If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest *interesting* programming language...
  - Turing complete
  - higher order (functions as data)
- Indeed, in the lambda-calculus, *all* computation happens by means of function abstraction and application.
- The *e. coli* of programming language research
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

$$\texttt{plus3} \quad = \quad \lambda\texttt{x. succ (succ (succ x))}$$

This function exists independent of the name `plus3`.

λ`x. t` is written "`fun x` $\rightarrow$ `t`" in OCaml.

So `plus3 (succ 0)` is just a convenient shorthand for "the function that, given `x`, yields `succ (succ (succ x))`, applied to `succ 0`."

$$\texttt{plus3 (succ 0)}$$
$$=$$
$$(\lambda\texttt{x. succ (succ (succ x))) (succ 0)}$$

Abstractions over Functions

Consider the λ-abstraction

$$\texttt{g} \quad = \quad \lambda\texttt{f. f (f (succ 0))}$$

Note that the parameter variable `f` is used in the *function* position in the body of `g`. Terms like `g` are called *higher-order* functions. If we apply `g` to an argument like `plus3`, the "substitution rule" yields a nontrivial computation:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{g plus3}} \\
= & (\lambda\texttt{f. f (f (succ 0))) (}\lambda\texttt{x. succ (succ (succ x)))} \\
\textit{i.e.} & (\lambda\texttt{x. succ (succ (succ x)))} \\
 & \quad ((\lambda\texttt{x. succ (succ (succ x))) (succ 0))} \\
\textit{i.e.} & (\lambda\texttt{x. succ (succ (succ x)))} \\
 & \quad \texttt{(succ (succ (succ (succ 0))))} \\
\textit{i.e.} & \texttt{succ (succ (succ (succ (succ (succ (succ 0))))))}
\end{array}
$$

Abstractions Returning Functions

Consider the following variant of `g`:

$$\texttt{double} \quad = \quad \lambda\texttt{f. }\lambda\texttt{y. f (f y)}$$

I.e., `double` is the function that, when applied to a function `f`, yields a *function* that, when applied to an argument `y`, yields `f (f y)`.

Example

```
       double plus3 0
=      (λf. λy. f (f y))
         (λx. succ (succ (succ x)))
         0
i.e.   (λy. (λx. succ (succ (succ x)))
               ((λx. succ (succ (succ x))) y))
         0
i.e.   (λx. succ (succ (succ x)))
               ((λx. succ (succ (succ x))) 0)
i.e.   (λx. succ (succ (succ x)))
               (succ (succ (succ 0)))
i.e.   succ (succ (succ (succ (succ (succ 0)))))
```

The Pure Lambda-Calculus

As the preceding examples suggest, once we have λ-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the "pure lambda-calculus"— *everything* is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function

Formalities

Syntax

$\mathtt{t}$ ::=		*terms*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x.t}$	*abstraction*
	$\mathtt{t\ t}$	*application*

*Term*inology:

- terms in the pure λ-calculus are often called λ-*terms*
- terms of the form $\lambda\mathtt{x.\ t}$ are called λ-*abstractions* or just *abstractions*

Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left

  E.g., $t\ u\ v$ *means* $(t\ u)\ v$, *not* $t\ (u\ v)$

- Bodies of λ- abstractions extend as far to the right as possible

  E.g., $\lambda x.\ \lambda y.\ x\ y$ *means* $\lambda x.\ (\lambda y.\ x\ y)$, *not* $\lambda x.\ (\lambda y.\ x)\ y$

Scope

The λ-abstraction term $\lambda\texttt{x.t}$ *binds* the variable $\texttt{x}$.
The *scope* of this binding is the *body* $\texttt{t}$.
Occurrences of $\texttt{x}$ inside $\texttt{t}$ are said to be *bound* by the abstraction.
Occurrences of $\texttt{x}$ that are *not* within the scope of an abstraction binding $\texttt{x}$ are said to be *free*.

$$\lambda\texttt{x. }\lambda\texttt{y. x y z}$$

Scope

The λ-abstraction term λ`x.t` *binds* the variable `x`.
The *scope* of this binding is the *body* `t`.
Occurrences of `x` inside `t` are said to be *bound* by the abstraction.
Occurrences of `x` that are *not* within the scope of an abstraction binding `x` are said to be *free*.

$$\lambda\texttt{x. }\lambda\texttt{y. x y z}$$
$$\lambda\texttt{x. (}\lambda\texttt{y. z y) y}$$

Values

$\mathtt{v}$ ::=		*values*
	$\lambda\mathtt{x.t}$	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]\,t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_{12}*."*

Operational Semantics

Computation rule:

$$(\lambda\mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]\,t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_{12}*."*

Congruence rules:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

Terminology

A term of the form $\texttt{(}\lambda\texttt{x.t) v}$ — that is, a λ-abstraction applied to a *value* — is called a *redex* (short for "reducible expression").

Alternative evaluation strategies

Strictly speaking, the language we have defined is called the *pure, call-by-value lambda-calculus*.

The evaluation strategy we have chosen — *call by value* — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction