Programming in the Lambda-Calculus

Multiple arguments

Above, we wrote a function `double` that returns a function as an argument.

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

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.
In general, λ`x.` λ`y. t` is a function that, given a value `v` for `x`, yields a function that, given a value `u` for `y`, yields `t` with `v` in place of `x` and `u` in place of `y`.
That is, λ`x.` λ`y. t` is a two-argument function.

(Recall the discussion of *currying* in OCaml.)

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* $\lambda x.\ (\lambda y.\ x)\ y$

The "Church Booleans"

$$\begin{array}{lcl} \texttt{tru} & = & \lambda\texttt{t.}\ \lambda\texttt{f.}\ \texttt{t} \\ \texttt{fls} & = & \lambda\texttt{t.}\ \lambda\texttt{f.}\ \texttt{f} \end{array}$$

$$\begin{array}{cll} & \texttt{tru v w} & \\ = & \underline{(\lambda\texttt{t.}\lambda\texttt{f.t})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow & \underline{(\lambda\texttt{f.}\ \texttt{v})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow & \texttt{v} & \text{reducing the underlined redex} \end{array}$$

$$\begin{array}{cll} & \texttt{fls v w} & \\ = & \underline{(\lambda\texttt{t.}\lambda\texttt{f.f})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow & \underline{(\lambda\texttt{f.}\ \texttt{f})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow & \texttt{w} & \text{reducing the underlined redex} \end{array}$$

Functions on Booleans

$$\texttt{not} \quad = \quad \lambda\texttt{b. b fls tru}$$

That is, `not` is a function that, given a boolean value `v`, returns `fls` if `v` is `tru` and `tru` if `v` is `fls`.

Functions on Booleans

$$\texttt{and} \quad = \quad \lambda\texttt{b.}\ \lambda\texttt{c.}\ \texttt{b c fls}$$

That is, `and` is a function that, given two boolean values `v` and `w`, returns `w` if `v` is `tru` and `fls` if `v` is `fls`

Thus `and v w` yields `tru` if both `v` and `w` are `tru` and `fls` if either `v` or `w` is `fls`.

Pairs

```
pair = λf.λs.λb. b f s
fst = λp. p tru
snd = λp. p fls
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.
By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean.

Example

$$
\begin{array}{lll}
 & \mathtt{fst\ (pair\ v\ w)} & \\
= & \mathtt{fst\ (}\underline{\mathtt{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ v}}\mathtt{\ w)} & \text{by definition} \\
\longrightarrow & \mathtt{fst\ (}\underline{\mathtt{(\lambda s.\ \lambda b.\ b\ v\ s)\ w}}\mathtt{)} & \text{reducing} \\
\longrightarrow & \mathtt{fst\ (\lambda b.\ b\ v\ w)} & \text{reducing} \\
= & \underline{\mathtt{(\lambda p.\ p\ tru)\ (\lambda b.\ b\ v\ w)}} & \text{by definition} \\
\longrightarrow & \underline{\mathtt{(\lambda b.\ b\ v\ w)\ tru}} & \text{reducing} \\
\longrightarrow & \mathtt{tru\ v\ w} & \text{reducing} \\
\longrightarrow^{*} & \mathtt{v} & \text{as before.}
\end{array}
$$

Church numerals

Idea: represent the number n by a function that "repeats some action n times."

$$
\begin{aligned}
\mathtt{c}_0 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{z} \\
\mathtt{c}_1 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ z} \\
\mathtt{c}_2 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ z)} \\
\mathtt{c}_3 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s\ (s\ (s\ z))}
\end{aligned}
$$

That is, each number n is represented by a term $\mathtt{c}_n$ that takes two arguments, $\mathtt{s}$ and $\mathtt{z}$ (for "successor" and "zero"), and applies $\mathtt{s}$, n times, to $\mathtt{z}$.

Functions on Church Numerals

Successor:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

$$\texttt{times = }\lambda\texttt{m. }\lambda\texttt{n. m (plus n) c}_0$$

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

times = λm. λn. m (plus n) c_0

Zero test:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

times = λm. λn. m (plus n) c_0

Zero test:

```
iszro = λm. m (λx. fls) tru
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

$\texttt{times = } \lambda\texttt{m. } \lambda\texttt{n. m (plus n) } \texttt{c}_0$

Zero test:

```
iszro = λm. m (λx. fls) tru
```

What about predecessor?

Predecessor

```
zz = pair c₀ c₀

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```

Normal forms

Recall:

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?
Prove it.

Normal forms

Recall:

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?
Prove it.
Does every term evaluate to a normal form?
Prove it.

Divergence

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

Note that `omega` evaluates in one step to itself!
So evaluation of `omega` never reaches a normal form: it *diverges*.

Divergence

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

Note that `omega` evaluates in one step to itself!
So evaluation of `omega` never reaches a normal form: it *diverges*.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of `omega` that are *very* useful...