CIS 500

Software Foundations

Fall 2002

25 September

The Pure Lambda Calculus

Syntax

t ::=		terms:
	x	variable
	λx.t	abstraction
	t t	application

Values

v ::=		values:
	λx.t	abstraction value

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

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

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

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

Congruence rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Terminology

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

Alternative evaluation strategies

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

Some other common ones:

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

Programming in the Lambda-Calculus

Multiple arguments

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

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

This idiom — a λ-abstraction that does *nothing* but immediately yield another abstraction — is very common in the λ-calculus.

In general, $\lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{t}$ is a function that, given a value $\texttt{v}$ for $\texttt{x}$, yields a function that, given a value $\texttt{u}$ for $\texttt{y}$, yields $\texttt{t}$ with $\texttt{v}$ in place of $\texttt{x}$ and $\texttt{u}$ in place of $\texttt{y}$.

That is, $\lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{t}$ is a two-argument function.

Aside: Currying

The transformation from a function taking a pair of arguments (in a language like OCaml that provides pairs) to a one-argument function returning another one-argument function is called currying.

It is considered *good* style in OCaml to define functions in curried style whenever possible.

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

The "Church Booleans"

```
tru  =  λt. λf. t
fls  =  λt. λf. f
```

```
        tru v w
   =    (λt.λf.t) v w     by definition
  ⟶    (λf. v) w          reducing the underlined redex
  ⟶    v                  reducing the underlined redex

        fls v w
   =    (λt.λf.f) v w     by definition
  ⟶    (λf. f) w          reducing the underlined redex
  ⟶    w                  reducing the underlined redex
```


Functions on Booleans

```
not   =   λb. 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

```
and   =   λb. λc. 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

	`fst (pair v w)`	
=	`fst ((λf. λs. λb. b f s) v w)`	by definition
⟶	`fst ((λs. λb. b v s) w)`	reducing the underlined redex
⟶	`fst (λb. b v w)`	reducing the underlined redex
=	`(λp. p tru) (λb. b v w)`	by definition
⟶	`(λb. b v w) tru`	reducing the underlined redex
⟶	`tru v w`	reducing the underlined redex
$\longrightarrow^*$	`v`	as before.

Church numerals

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

c_0 = λs. λz. z
c_1 = λs. λz. s z
c_2 = λs. λz. s (s z)
c_3 = λs. λz. s (s (s z))

That is, each number n is represented by a term c_n that takes two arguments, s and z (for "successor" and "zero"), and applies s, n times, to 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:

times = λm. λn. 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) c0
```

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) c0
```

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:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
iszro = λm. m (λx. fls) tru
```

What about predecessor?

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))
```

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```


Normal forms

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

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

```
omega  =  (λx. x x) (λx. x x)
```

Note that omega evaluates in *one* step to itself!

So evaluation of omega never reaches a normal form: it diverges.

Divergence

```
omega  =  (λx. x x) (λx. 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...

Iterated Application

Suppose `f` is some λ-abstraction, and consider the following term:

$$Y_f \;=\; (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Iterated Application

Suppose `f` is some λ-abstraction, and consider the following term:

$$Y_f \;=\; (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Now the "pattern of divergence" becomes more interesting:

$$\begin{array}{c} Y_f \\ = \\ \underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))} \\ \longrightarrow \\ f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}) \\ \longrightarrow \\ f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})) \\ \longrightarrow \\ f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))) \\ \longrightarrow \\ \cdots \end{array}$$

Y_f is still not very useful, since (like `omega`), all it does is diverge.

Is there any way we could "slow it down"?

Delaying Divergence

```
poisonpill  =  λy. omega
```

Note that `poisonpill` is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

```
(λp. fst (pair p fls) tru) poisonpill
⟶
fst (pair poisonpill fls) tru
⟶*
poisonpill tru
⟶
omega
⟶
...
```


A delayed variant of `omega`

Here is a variant of `omega` in which the delay and divergence are a bit more tightly intertwined:

```
omegav  =  λy. (λx. (λy. x x y)) (λx. (λy. x x y)) y
```

Note that `omegav` is a normal form. However, if we apply it to any argument `v`, it diverges:

```
omegav v
=
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
⟶
(λx. (λy. x x y)) (λx. (λy. x x y)) v
⟶
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
=
omegav v
```


Another delayed variant

Suppose `f` is a function. Define

$$Z_f \;=\; \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y$$

This term combines the "added `f`" from Y_f with the "delayed divergence" of `omegav`.

If we now apply Z_f to an argument `v`, something interesting happens:

$$Z_f\ v$$
$$=$$
$$(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v$$
$$\longrightarrow$$
$$(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ v$$
$$\longrightarrow$$
$$f\ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v$$
$$=$$
$$f\ Z_f\ v$$

Since Z_f and `v` are both values, the next computation step will be the reduction of `f` Z_f — that is, before we "diverge," `f` gets to do some computation.

Now we are getting somewhere.

Recursion

Let

```
f  =  λfct.
        λn.
          if n=0 then 1
          else n * (fct (pred n))
```

`f` looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function `fct`, which is passed as a parameter.

N.b.: for brevity, this example uses "real" numbers and booleans, infix syntax, etc...

We can use `Z` to "tie the knot" in the definition of `f` and obtain a real recursive factorial function:

$$
\begin{array}{c}
\mathtt{Z_f\ 3} \\
\longrightarrow^* \\
\mathtt{f\ Z_f\ 3} \\
= \\
\mathtt{(\lambda fct.\ \lambda n.\ \ldots)\ Z_f\ 3} \\
\longrightarrow\ \longrightarrow \\
\mathtt{if\ 3{=}0\ then\ 1\ else\ 3 * (Z_f\ (pred\ 3))} \\
\longrightarrow^* \\
\mathtt{3 * (Z_f\ (pred\ 3)))} \\
\longrightarrow \\
\mathtt{3 * (Z_f\ 2)} \\
\longrightarrow^* \\
\mathtt{3 * (f\ Z_f\ 2)} \\
\ldots
\end{array}
$$

A Generic Z

If we define

$$\mathtt{Z} = \lambda\mathtt{f}.\ \mathtt{Z_f}$$

i.e.,

$$\mathtt{Z} = \lambda\mathtt{f}.\ \lambda\mathtt{y}.\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))\ \mathtt{y}$$

then we can obtain the behavior of $\mathtt{Z_f}$ for any `f` we like, simply by applying `Z` to `f`.

$$\mathtt{Z\ f} \longrightarrow \mathtt{Z_f}$$

N.b.:

The term `Z` here is essentially the same as the `fix` discussed the book.

$$\mathtt{Z} = \lambda\mathtt{f}.\ \lambda\mathtt{y}.\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))\ \mathtt{y}$$

$$\mathtt{fix} = \lambda\mathtt{f}.\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))\ (\lambda\mathtt{x}.\ \mathtt{f}\ (\lambda\mathtt{y}.\ \mathtt{x\ x\ y}))$$

`Z` is hopefully slightly easier to understand, since it has the property that $\mathtt{Z\ f\ v} \longrightarrow^* \mathtt{f\ (Z\ f)\ v}$, which `fix` does not (quite) share.