CIS 500

Software Foundations

Fall 2002

25 September

The Pure Lambda Calculus

Syntax

t	::=	terms:
	x	variable
	$\lambda x.t$	abstraction
	$t\ t$	application

Values

v	::=	values:
	$\lambda x.t$	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 `(λx.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.

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

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 `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, $\lambda \texttt{x}.\ \lambda \texttt{y}.\ \texttt{t}$ 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

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

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

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

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 (`$\underline{\texttt{(}\lambda\texttt{f. }\lambda\texttt{s. }\lambda\texttt{b. b f s) v}}$` w)`	by definition
$\longrightarrow$	`fst (`$\underline{\texttt{(}\lambda\texttt{s. }\lambda\texttt{b. b v s) w}}$`)`	reducing the underlined redex
$\longrightarrow$	`fst (`λ`b. b v w)`	reducing the underlined redex
$=$	$\underline{\texttt{(}\lambda\texttt{p. p tru) (}\lambda\texttt{b. b v w)}}$	by definition
$\longrightarrow$	$\underline{\texttt{(}\lambda\texttt{b. b v w) tru}}$	reducing the underlined redex
$\longrightarrow$	`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) c0
```

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:

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

Iterated Application

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

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

Now the "pattern of divergence" becomes more interesting:

$$Y_f$$
$$=$$
$$\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}$$
$$\longrightarrow$$
$$f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})$$
$$\longrightarrow$$
$$f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))$$
$$\longrightarrow$$
$$f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})))$$
$$\longrightarrow$$
$$\ldots$$

`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

$$\texttt{poisonpill} \quad = \quad \lambda\texttt{y. omega}$$

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.

$$\underline{(\lambda\texttt{p. fst (pair p fls) tru) poisonpill}}$$
$$\longrightarrow$$
$$\texttt{fst (pair poisonpill fls) tru}$$
$$\longrightarrow^*$$
$$\underline{\texttt{poisonpill tru}}$$
$$\longrightarrow$$
$$\texttt{omega}$$
$$\longrightarrow$$
$$\ldots$$

A delayed variant of omega

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

$$\texttt{omegav} \;=\; \lambda y.\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y$$

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

$$\texttt{omegav v}$$
$$=$$
$$\underline{(\lambda y.\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y)\ v}$$
$$\longrightarrow$$
$$\underline{(\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))}\ v$$
$$\longrightarrow$$
$$(\lambda y.\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y)\ v$$
$$=$$
$$\texttt{omegav v}$$

Another delayed variant

Suppose `f` is a function. Define

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

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:

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

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:

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

A Generic Z

If we define

$$Z \quad = \quad \lambda f.\ Z_f$$

i.e.,

$$Z \quad = \quad \lambda f.\ \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y$$

then we can obtain the behavior of Z_f for any f we like, simply by applying Z to f.

$$Z\ f \quad \longrightarrow \quad Z_f$$

N.b.:

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

```
Z   =  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
fix =  λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

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