Chapter 5: The Untyped Lambda Calculus

What is lambda calculus for?

Basics: syntax and operational semantics

Programming in the Lambda Calculus

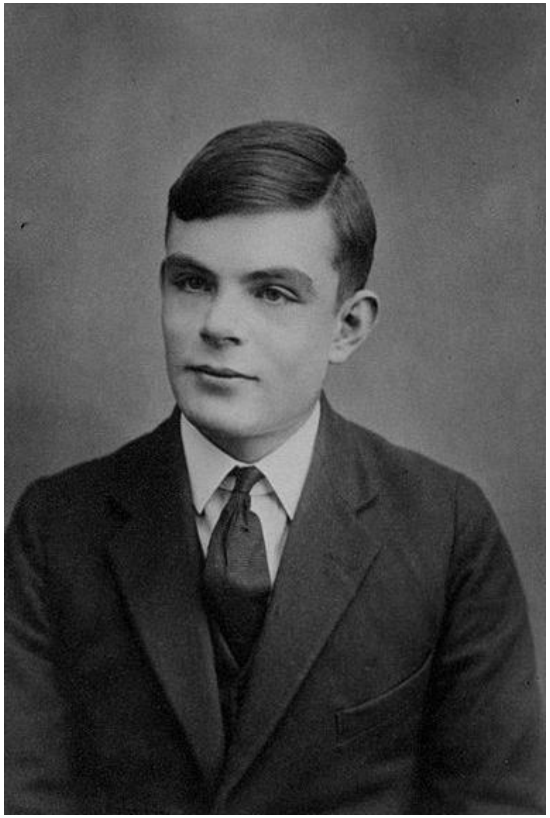
Formalities (formal definitions)

Review

- Core messages in the previous lecture
  - (Untyped) programming languages are defined by syntax and semantics
  - Syntax is often specified by grammars
  - Semantics can be specified in three ways, and this book chooses operational semantics expressed as evaluation rules
  - Big step vs small step semantics

Story of Turing and Church

Alonzo Church
Lambda Calculus



Alan Turing
Turing Machine

What is Lambda calculus for?

- A core calculus (used by Landin) for
  - capturing the language's essential mechanisms,
  - with a collection of convenient derived forms whose behavior is understood by translating them into the core

- A formal system invented in the 1920s by Alonzo Church (1936, 1941), in which all computation is reduced to the basic operations of function definition and application.

Basics

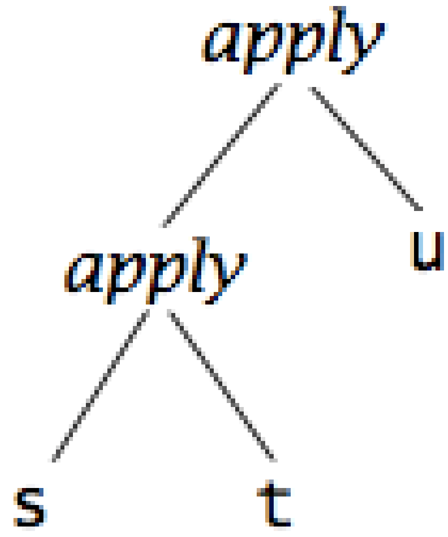
Syntax

- The lambda-calculus (or λ-calculus) embodies this kind of function definition and application in the purest possible form.

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

Abstract Syntax Trees

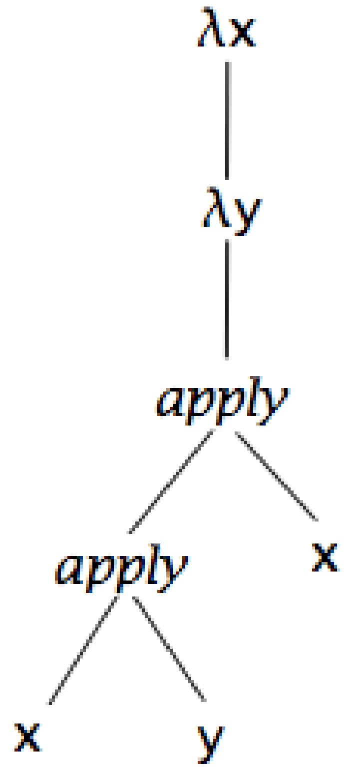
- (s t) u   (or simply written as s t u)



Abstract Syntax Trees

- λx. (λy. ((x y) x))

  (or simply written as λx. λy. x y x)



Scope

- An occurrence of the variable x is said to be bound when it occurs in the body t of an abstraction λx.t.
  - λx is a binder whose scope is t. A binder can be renamed: e.g., λx.x = λy.y.
  - So-called: alpha-renaming

- An occurrence of x is free if it appears in a position where it is not bound by an enclosing abstraction on x.
  - **Exercises**: Find free variable occurrences from the following terms: x y, λx.x, λy. x y, (λx.x) x.

Operational Semantics

- Beta-reduction: the only computation

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

"the term obtained by replacing all free occurrences of x in t_{12} by t_2 "

A term of the form (λx.t12) t2 is called a redex.

- Examples:

(λx.x) y → y

(λx. x (λx.x)) (u r) → u r (λx.x)

Evaluation Strategies

- Full beta-reduction
  - Any redex may be reduced at any time.

- Example:
  - Let id = λx.x. We can apply beta reduction to any of the following underlined redexes:

$$\underline{\mathtt{id\ (id\ (\lambda z.\ id\ z))}}$$

$$\mathtt{id\ (}\underline{\mathtt{(id\ (\lambda z.\ id\ z))}}\mathtt{)}$$

$$\mathtt{id\ (id\ (\lambda z.\ }\underline{\mathtt{id\ z}}\mathtt{))}$$

Note: lambda calculus is confluent under full beta-reduction.
Ref. Church-Rosser property.

Evaluation Strategies

- The normal order strategy
  - The leftmost, outmost redex is always reduced first.

$$
\begin{array}{ll}
 & \underline{\mathtt{id}\ (\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))} \\
\longrightarrow & \underline{\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z)} \\
\longrightarrow & \lambda z.\ \underline{\mathtt{id}\ z} \\
\longrightarrow & \lambda z.z \\
\not\longrightarrow &
\end{array}
$$

Evaluation Strategies

- The call-by-name strategy
  - A more restrictive normal order strategy, allowing no reduction inside abstraction.

$$\begin{array}{ll} & \underline{\mathtt{id}\ (\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))} \\ \longrightarrow & \underline{\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z)} \\ \longrightarrow & \lambda z.\ \mathtt{id}\ z \\ \not\longrightarrow & \end{array}$$

Evaluation Strategies

- The call-by-value strategy
  - only outermost redexes are reduced and where a redex is reduced only when its right-hand side has already been reduced to a value
  - Value: a term that cannot be reduced any more.

$$
\begin{array}{ll}
 & \mathtt{id}\ \underline{(\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z))} \\
\longrightarrow & \underline{\mathtt{id}\ (\lambda z.\ \mathtt{id}\ z)} \\
\longrightarrow & \lambda z.\ \mathtt{id}\ z \\
\not\longrightarrow &
\end{array}
$$

Programming in the Lambda Calculus

Multiple Arguments

Church Booleans

Pairs

Church Numerals

Recursion

Multiple Arguments

$f\ (x, y) = s$

currying

$(f\ x)\ y = s$

$f = \lambda x.\ (\lambda y.\ s)$

Church Booleans

- Boolean values can be encoded as:

  tru = λt. λf. t

  fls = λt. λf. f

- Boolean conditional and operators can be encoded as:

  test = λl. λm. λn. l m n

  and = λb. λc. b c fls

Church Booleans

- An Example

$$
\begin{array}{ll}
 & \mathtt{test\ tru\ v\ w} \\
= & \underline{\mathtt{(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ tru}}\ \mathtt{v\ w} \\
\longrightarrow & \underline{\mathtt{(\lambda m.\ \lambda n.\ tru\ m\ n)\ v}}\ \mathtt{w} \\
\longrightarrow & \underline{\mathtt{(\lambda n.\ tru\ v\ n)\ w}} \\
\longrightarrow & \mathtt{tru\ v\ w} \\
= & \underline{\mathtt{(\lambda t.\lambda f.t)\ v}}\ \mathtt{w} \\
\longrightarrow & \underline{\mathtt{(\lambda f.\ v)\ w}} \\
\longrightarrow & \mathtt{v}
\end{array}
$$

Church Booleans

- Can you define *or*?

- $or = \lambda a. \lambda b. a\ tru\ b$

Church Numerals

- Encoding Church numerals:

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
etc.
```

- Defining functions on Church numerals:

succ = λn. λs. λz. s (n s z);

plus = λm. λn. λs. λz. m s (n s z);

times = λm. λn. m (plus n) c0;

Church Numerals

- Can you define minus?

- Suppose we have pred, can you define minus?
  - $\lambda m.\lambda n.n\ pred\ m$

- Can you define pred?
  - $\lambda n.\lambda s.\lambda z.n\ \big(\lambda g.\lambda h.h\ (g\ s)\big)\ (\lambda u.z)\ (\lambda u.u)$
  - Basic idea: skipping the last application of s
  - $(\lambda u.z)$ -- a wrapped zero
  - $(\lambda u.u)$ – the last application to be skipped
  - $\big(\lambda g.\lambda h.h\ (g\ s)\big)$ -- apply h if it is the last application, otherwise apply g
  - Try n = 0, 1, 2 to see the effect

Pairs

- Encoding

$$
\begin{aligned}
&\mathtt{pair} = \lambda f.\lambda s.\lambda b.\ b\ f\ s;\\
&\mathtt{fst} = \lambda p.\ p\ \mathtt{tru};\\
&\mathtt{snd} = \lambda p.\ p\ \mathtt{fls};
\end{aligned}
$$

- An Example

$$
\begin{array}{ll}
 & \mathtt{fst}\ (\mathtt{pair}\ v\ w)\\
= & \mathtt{fst}\ (\underline{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ v}\ w)\\
\longrightarrow & \mathtt{fst}\ (\underline{(\lambda s.\ \lambda b.\ b\ v\ s)\ w})\\
\longrightarrow & \mathtt{fst}\ (\lambda b.\ b\ v\ w)\\
= & \underline{(\lambda p.\ p\ \mathtt{tru})\ (\lambda b.\ b\ v\ w)}\\
\longrightarrow & \underline{(\lambda b.\ b\ v\ w)\ \mathtt{tru}}\\
\longrightarrow & \mathtt{tru}\ v\ w\\
\longrightarrow^{*} & v
\end{array}
$$

Recursion

- Terms with no normal form are said to diverge.

  omega = (λx. x x) (λx. x x);

- Fixed-point combinator

  fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));

  Note: fix f = f (λy. (fix f) y)

Recursion

- Basic Idea:

A recursive definition: h = <body containing h>

g = λf . <body containing f>

h = fix g

Recursion

- Example:

  fac = λn. if eq n c0

  then c1

  else times n (fac (pred n))

  ⇩

  g = λf . λn. if eq n c0

  then c1

  else times n (f (pred n))

  fac = fix g

Exercise: Check that fac c3 → c6.

Y Combinator

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

$$fix = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

- Y f = f (Y f)
- Why fix is used instead of Y?

Answer

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

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

- Assuming call-by-value
  - `(x x)` is not a value
  - while `(λy. x x y)` is
  - Y will diverge for any f

Formalities (Formal Definitions)

Syntax (free variables)

Substitution

Operational Semantics

Syntax

- **Definition** [Terms]: Let V be a countable set of variable names. The set of terms is the smallest set T such that

1. $x \in T$ for every $x \in V$;
2. if $t_1 \in T$ and $x \in V$, then $\lambda x.t_1 \in T$;
3. If t1 $\in$ T and $t_2 \in T$, then $t_1\ t_2 \in T$.

- Free Variables

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$$

Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}
$$

Alpha-conversion: Terms that differ only in the names of bound variables are interchangeable in all contexts.

Example:

$$
\begin{aligned}
& [x \rightarrow y\ z]\ (\lambda y.\ x\ y) \\
=\ & [x \rightarrow y\ z]\ (\lambda w.\ x\ w) \\
=\ & \lambda w.\ y\ z\ w
\end{aligned}
$$

Operational Semantics

Syntax

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

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

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

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

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

Summary

- What is lambda calculus for?
  - A core calculus for capturing language essential mechanisms
  - Simple but powerful
- Syntax
  - Function definition + function application
  - Binder, scope, free variables
- Operational semantics
  - Substitution
  - Evaluation strategies: normal order, call-by-name, call-by-value

Homework

- Understand Chapter 5.
- Do exercise 5.3.6 in Chapter 5.

5.3.6 EXERCISE [★★]: Adapt these rules to describe the other three strategies for evaluation—full beta-reduction, normal-order, and lazy evaluation. □