编程语言的设计原理

Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2025

Recap

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

Abstract Machines

- An abstract machine consists of:
  - a set of *states*
  - a *transition relation* on states, written as $\longrightarrow$

    "$t \longrightarrow t'$ " is read as "t evaluates to t' in *one step*".

- A *state* records all the information in the abstract machine at a given moment.
  - e.g., an abstract-machine-style description of a conventional microprocessor would include the program counter, the contents of the registers, the contents of main memory, and the machine code program being executed.

Operational semantics for Booleans

- Syntax of terms and values

`t`	::=		*terms*
		`true`	*constant true*
		`false`	*constant false*
		`if t then t else t`	*conditional*
`v`	::=		*values*
		`true`	*true value*
		`false`	*false value*

Evaluation relation for Booleans

- The evaluation relation $t \longrightarrow t'$ is **the smallest relation closed** under the following rules:

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2 \quad \text{(E-IfTrue)}$$

$$\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3 \quad \text{(E-IfFalse)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \quad \text{(E-If)}$$

Evaluation relation for Boolean

- Computation rules

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \quad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \quad \text{(E-IfFalse)}$$

- Congruence rules

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-If)}$$

- Computation rules perform *"real" computation* steps
- Congruence rules determine *where* *computation rules* can be *applied* next

Evaluation relation for Booleans

$\longrightarrow$ is the *smallest two-place relation* closed under the following rules:

$$((\texttt{if true then } t_2 \texttt{ else } t_3), t_2) \in \longrightarrow$$

$$((\texttt{if false then } t_2 \texttt{ else } t_3), t_3) \in \longrightarrow$$

$$\frac{(t_1, t_1') \in \longrightarrow}{((\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3), (\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3)) \in \longrightarrow}$$

The notation $t \longrightarrow t'$ is short-hand for $(t, t') \in \longrightarrow$.

If the pair (t, t') is an evaluation relation, then the evaluation statement or judgement $t \longrightarrow t'$ is said to be derivable

Derivation

- "**Justification**" for ***a particular pair of terms*** that are in the evaluation relation in *the form of a tree*.

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \textsc{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \textsc{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \textsc{E-If}$$

  – These trees are called *derivation trees* (or just *derivations*).
  – The **final statement** in a derivation is its **conclusion**.
  – We say that the derivation is a **witness** for its conclusion (or a **proof** of its conclusion) — it records *all the reasoning steps* that justify the conclusion.

Induction on Derivation

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \text{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \text{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \text{E-If}$$

- Write proofs about evaluation "*by induction on derivation trees*."
- Given an arbitrary derivation $\mathcal{D}$ with conclusion $t \longrightarrow t'$, we assume the desired result for its *immediate sub-derivation* (if any) and proceed by *a case analysis* of *the final evaluation rule* used in constructing the derivation tree.

Induction on Derivation

Theorem [Determinacy of one-step evaluation]:

If $t \to t'$. and $t \to t''$, then $t' = t''$.

Proof. By induction on derivation of $t \to t'$.

If *the last rule* used in the derivation of $t \to t'$ is E-IfTrue, then t has the form

if true then t2 else t3.

It can be shown that there is only one way to reduce such t.

......

Normal Form

Definition: A term *t* is in normal form if *no evaluation rule* applies to it.

Theorem: Every *value* is in normal form.

Theorem: If *t* is in normal form, then *t* is a *value*.

Prove by contradiction (then by structural induction).

Multi-step Evaluation Relation

Definition: The multi-step evaluation relation $\longrightarrow^*$ is the *reflexive*, *transitive closure* of one-step evaluation.

Theorem [Uniqueness of normal forms]:

If t $\longrightarrow^*$ u and t $\longrightarrow^*$ u′, where u and u′ are both normal forms, then u = u′.

Theorem [Termination of Evaluation]:

For every term t there is some normal form t′ such that t $\longrightarrow^*$ t′.

Extending Evaluation to Numbers

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\text{succ } t_1 \longrightarrow \text{succ } t_1'} \quad \text{(E-Succ)}$$

$$\text{pred } 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\text{pred (succ } nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{pred } t_1 \longrightarrow \text{pred } t_1'} \quad \text{(E-Pred)}$$

$$\text{iszero } 0 \longrightarrow \text{true} \quad \text{(E-IszeroZero)}$$

$$\text{iszero (succ } nv_1) \longrightarrow \text{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{iszero } t_1 \longrightarrow \text{iszero } t_1'} \quad \text{(E-IsZero)}$$

Big-step Evaluation

$$v \Downarrow v \qquad \text{(B-VALUE)}$$

$$\frac{t_1 \Downarrow \mathtt{true} \qquad t_2 \Downarrow v_2}{\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \Downarrow v_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{t_1 \Downarrow \mathtt{false} \qquad t_3 \Downarrow v_3}{\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \Downarrow v_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{t_1 \Downarrow nv_1}{\mathtt{succ}\ t_1 \Downarrow \mathtt{succ}\ nv_1} \qquad \text{(B-SUCC)}$$

$$\frac{t_1 \Downarrow 0}{\mathtt{pred}\ t_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{t_1 \Downarrow \mathtt{succ}\ nv_1}{\mathtt{pred}\ t_1 \Downarrow nv_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{t_1 \Downarrow 0}{\mathtt{iszero}\ t_1 \Downarrow \mathtt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{t_1 \Downarrow \mathtt{succ}\ nv_1}{\mathtt{iszero}\ t_1 \Downarrow \mathtt{false}} \qquad \text{(B-ISZEROSUCC)}$$

Stuckness

Definition: A closed term is stuck if it is in *normal form* but *not a value*.

Examples:

- succ true
- succ false
- if zero then true else false

Summary

- How to define syntax?
  - Grammar, Inductively, Inference Rules, Generative

- How to define semantics?
  - Operational, Denotational, Axomatic

- How to define evaluation relation (operational semantics)?
  - Small-step/Big-step evaluation relation
  - Normal form
  - Confluence/termination

Chapter 5
The Untyped Lambda Calculus

What is lambda calculus for ?

Basics: Syntax and Operational semantics

Programming in the Lambda Calculus

Formalities (formal definitions)

Why Lambda calculus?

- 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))

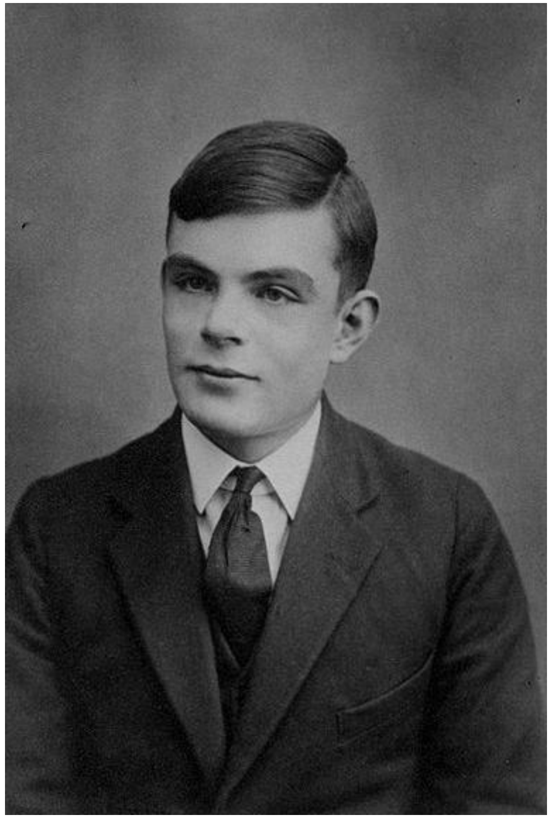
i.e., plus3 x is succ (succ (succ x))

Q: What is plus3 itself?

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

Story of Turing and Church

Alonzo Church
Lambda Calculus
lambda definable
Church' thesis



Alan Turing
Turing Machine
Turing computability

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.
  - modeling programming language, as the foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...) , and being *central to contemporary computer science*.

Lambda calculus

- A formal system devised by Alonzo Church in the 1930's as a model for computability
  - *all computation* is reduced to the *basic operations of function abstraction and application*.
- A very simple but very powerful language based on pure abstraction, with
  - Turing complete
  - Higher order (functions as data)

Lambda calculus

- Widely used in the specification of programming language features, in language design and implementation, and in the study of type systems
- Important due to ***the fact*** that it can be viewed simultaneously as
  – *a simple programming language* in which computations can be described and
  – *a mathematical object* about which rigorous statements can be proved
- Can be enriched in a variety of ways

Basics

Syntax

Scope

Operational semantics

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*

- Terminology:
  – terms in the pure λ-calculus are often called λ-*terms*
  – terms of the form λx. t are called λ-*abstractions* or just abstractions

Syntax

- Recall the function

plus3 x = succ (succ (succ x))

- Write it with λ-*terms* as:

plus3 = λx. succ (succ (succ x))

Note:

This function exists independent of the name plus3

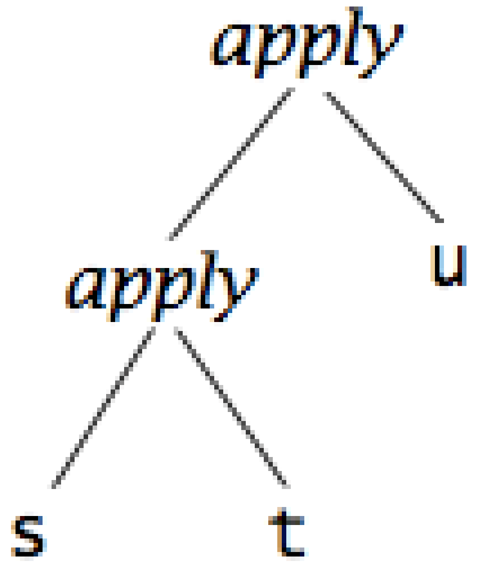
λx.t is written "fun x → t" in OCaml.

Abstract and Concrete Syntax

- It is useful to distinguish **the syntax of programming languages** at *two levels of structure*:
  - ***Concrete syntax*** (or surface syntax) of the language refers to the *strings of characters* that programmers directly read and write
  - ***Abstract syntax*** is a *much simpler internal representation* of programs as *labeled trees* (called *abstract syntax trees* or ASTs)
    - The tree representation renders **the structure of terms** immediately obvious, making it a natural fit for the complex manipulations involved in both rigorous language definitions (and proofs about them) and the internals of compilers and interpreters.

Abstract Syntax Trees

- (s t) u



Syntactic conventions

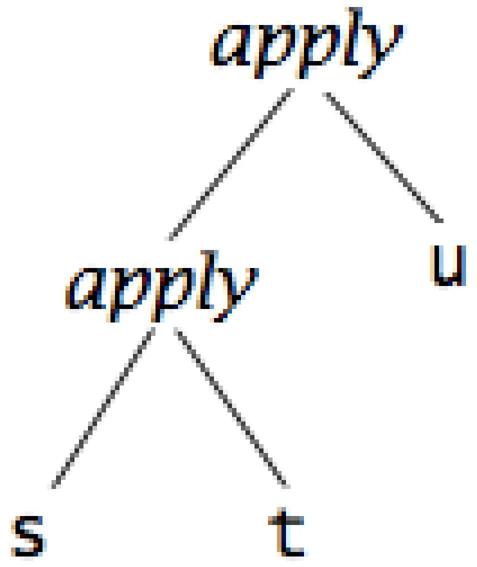
- The λ-calculus provides *only one-argument functions*, all multi-argument functions must be written in curried style.
- The following *conventions* make the linear forms of terms easier to read and write:
  - Application *associates to the left*

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

Abstract Syntax Trees

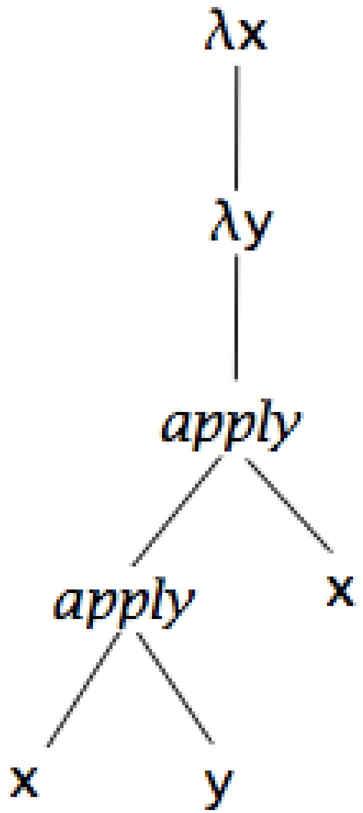
- (s t) u   (or simply written as s t u)



Abstract Syntax Trees

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

    (or simply written as $\lambda x.\ \lambda y.\ x\ y\ x$)



Scope

- *An occurrence* of the variable x is said to be ***bound*** when it occurs in the body t of an abstraction $\lambda \mathrm{x.t}$, i.e.,
  - the λ-abstraction term $\lambda x.\mathrm{t}$ binds the variable x , and the scope of this binding is the body t.
  - λx is a *binder* whose *scope* is t.
  - a binder can be *renamed* as necessary
    - so-called: *alpha-renaming*
    - e.g., $\lambda x.\mathrm{x} = \lambda \mathrm{y}.\ \mathrm{y}$

Scope

- An occurrence of x is *free* if it appears in a position where it is not bound *by an enclosing abstraction* on x.
  - a ***term with no free variable*** is said to be *closed*.
  - *closed terms* are also called *combinators*.
- **Exercises**: Find free variable occurrences from the following terms:
  - x y,
  - λx.x
  - λy.x y
  - (λx.x) x
  - (λx.x) (λy.y x)
  - (λx.x) (λx.x)
  - (λx.(λy.x y)) y

Operational Semantics

- If the function $\lambda x.t$ is applied to t_2, we ***substitute*** *all free occurrences of* x in t with t_2.
  - If the substitution would bring a free variable of t_2 in an expression *where this variable occurs bound*, we *rename the bound variable* before performing the substitution.

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

- Examples:

  $(\lambda x.x)\ (\lambda x.x) \rightarrow\ ?$

  $(\lambda x.(\lambda y.x\ y))\ y \rightarrow\ ?$

  $(\lambda x.(\lambda y.(x\ (\lambda x.x\ y))))\ y \rightarrow ?$

Operational Semantics

- *Beta-reduction*: the only computation (substitution)

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

  - the term obtained by *replacing all free occurrences* of x in t_{12} by t_2
  - a term of the form *(λx.t) v* — a *λ-abstraction* applied to a *value* — is called a *redex* (short for "*reducible expression*")
  - the operation of rewriting a *redex* according to the above rule is called *beta-reduction*
- Examples:

$$(\lambda x.\, x)\; y \rightarrow y$$

$$(\lambda x.\, x\; (\lambda x.\, x))\; (u\; r) \rightarrow u\; r\; (\lambda x.\, x)$$

Values

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

Evaluation Strategies

- Full beta-reduction
  - *any redex* may be reduced *at any time*.
- e. g., *id = λx.x*, consider

  (λx.x) ((λx.x) (λz. (λx.x) z))

  - we can apply *full beta reduction* to *any* of the following *underlined redexes*:

id (id (λz. id z))	outermost
id ((id (λz. id z)))	middle
id (id (λz. id z))	innermost

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*.
    - try to reduce always the leftmost expression of a series of applications, and continue until *no further reductions* are possible
  - the evaluation relation under this strategy is actually ***a partial function***: each term *t* evaluates in one step to **at most one** term *t'*

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

Evaluation Strategies

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

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

  - stop before the last and regard $\lambda z.\ \mathrm{id}\ z$ as a *normal form*
  - *call-by-need*

Evaluation Strategies

- *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.*

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

Evaluation Strategies

- *call-by-value* strategy
  - **strict** in the sense that *the arguments to functions are always evaluated*, *whether or not they are used* by the body of the function.
  - reflects standard conventions found in most mainstream languages.
  - adopted in our course

- The choice of evaluation strategy actually *makes little difference* when discussing type systems.
  - The issues that motivate various typing features, and the techniques used to address them, are much the same for all the strategies.

Evaluation Strategies: summary

- Full beta-reduction
  - *any redex* may be reduced *at any time*.
  - **confluent** under full beta-reduction
- normal order strategy
  - The *leftmost, outmost redex* is always reduced *first*.
- *call-by-name* strategy
  - a *more restrictive normal order* strategy, *allowing no reduction inside abstraction*.
- *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*
  - **strict** in the sense that *the arguments to functions are always evaluated*, *whether or not they are used* by the body of the function.
  - reflects standard conventions found in most mainstream languages.
  - adopted in our course

Operational Semantics

- Computation rule

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

- Congruence rules

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

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

Lambda Calculus

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

Abstractions over Functions

- Consider the λ-abstraction

$$g = \lambda f.\ f\ (f\ (succ\ 0))$$

  - 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:

```
g plus3
  =    (λf. f (f (succ 0))) (λx. succ (succ (succ x)))
  i.e. (λx. succ (succ (succ x)))
          ((λx. succ (succ (succ x))) (succ 0))
  i.e. (λx. succ (succ (succ x)))
          (succ (succ (succ (succ 0))))
  i.e. succ (succ (succ (succ (succ (succ (succ 0))))))
```

Programming in the Lambda Calculus

Multiple Arguments
Church Booleans
Pairs
Church Numerals
Recursion

Multiple Arguments

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

f (x, y) = t (i.e., f x y)

currying ↓

(f x) y = t

λ-encoding ↓

f = λx. (λy. t)

Multiple Arguments

- In general, $\lambda x.\ \lambda y.\ s$ 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.
  – i.e., $f = \lambda x.\ \lambda y.\ s$ is a *two-argument function*.
- Apply f to its arguments one at a time
  – e.g., f v w ⟺ (f v) w ⟺

$$(\lambda y.\ [x \mapsto v]\ s)\ w \iff [y \mapsto w]\ [x \mapsto v]\ s$$

- λ-abstraction that does nothing but *immediately yields another abstraction* — is very common in the λ-calculus.

Church Booleans

- Boolean values can be encoded as:

$$tru = \lambda t.\lambda f.t$$
$$fls = \lambda t.\lambda f.f$$

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

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

Church Booleans

- Boolean conditional and operators can be encoded as a combinator:

$$test = \lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n$$

	test tru v w	
=	(λl. λm. λn. l m n) tru v w	by definition
→	(λm. λn. tru m n) v w	reducing the underlined redex
→	(λn. tru v n) w	reducing the underlined redex
→	tru v w	reducing the underlined redex
=	(λt.λf.t) v w	by definition
→	(λf. v) w	reducing the underlined redex
→	v	reducing the underlined redex

Church Booleans

- How to define *not*?
  - a function that, given a boolean value v, returns fls if v is tru and tru if v is fls.

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

Church Booleans

- Boolean conditional
  - 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.

- and operators can be encoded as:

$$\text{and} = \lambda b.\ \lambda c.\ b\ c\ \text{fls}$$

Church Booleans

- How to define *or* ?

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

Pairs

- Encoding

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

- Example

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

Church Numerals

- *Encoding Church numerals*
  – *Basic* idea: represent the number n by ***a function*** that "repeats ***some action*** n *times*", making numbers into ***active entities***

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$

  – each number n is represented by *a term* c_n taking *two arguments*, s and z (for "successor" and "zero"), and applies s, n times, to z.

Functions on Church Numerals

- Successor

$$\text{scc} = \lambda n. \underline{\lambda s. \lambda z.}\ s\ (\underline{n\ s\ z});$$

- Addition

$$\text{plus} = \lambda m. \lambda n. \underline{\lambda s. \lambda z.}\ m\ s\ (\underline{n\ s\ z});$$

Both **scc** and **plus** take some Church numeral **(n** for **scc** and **m,n** for **plus**) and **yield another Church numeral** —i.e., **a function-** that accepts arguments **s** and **z**, applies s iteratively to z

Functions on Church Numerals

- Multiplication

$$\text{times} \;=\; \lambda m.\, \lambda n.\, m\ (\text{plus}\ n)\ c0;$$

based on **plus:** since plus takes its arguments one at a time, applying it to just one argument **n** **yields the function** that adds **n** to whatever argument given, which is passed to m and c0: apply the function that adds **n** to its argument, iterated **m** times, to zero

- Zero test

$$\text{iszro} = \lambda m.\, m\ (\lambda x.\, \text{fls})\ \text{tru}$$

$$\text{iszro c0 ?}$$

$$\text{iszro c1 ?}$$

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*?
  - $\lambda n. \lambda s. \lambda z. n\ \big(\lambda g. \lambda h. h\ (g\ s)\big)\ (\lambda u. z)\ (\lambda u. u)$
  - $(\lambda u. z)$ -- 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

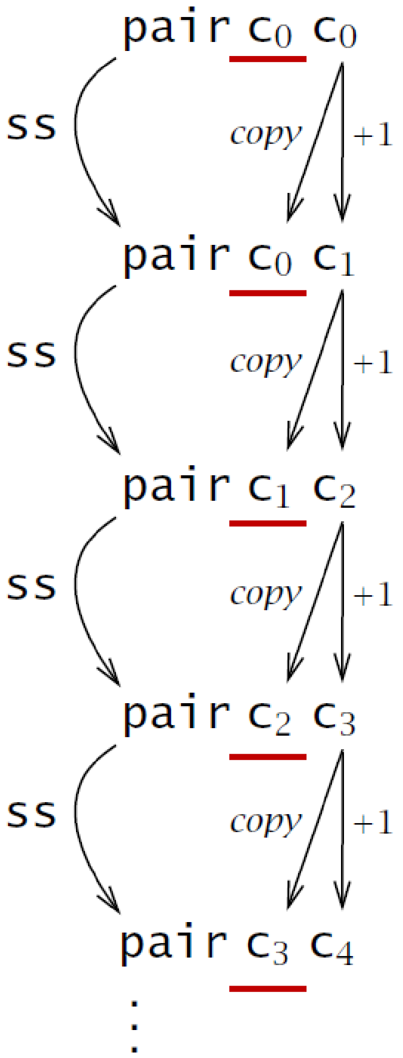
Church Numerals

- predecessor

$$zz = \text{pair}\ c_0\ c_0$$

$$ss = \lambda p.\ \text{pair}\ (\text{snd}\ p)\ (\text{scc}\ (\text{snd}\ p))$$

$$prd = \lambda m.\ \text{fst}\ (m\ ss\ zz)$$



Church Numerals

- We have seen that *booleans*, *numerals*, and the *operations on them* can be *encoded in the pure lambda-calculus* (λ).
- When working with examples, however, it is often convenient to include *the primitive booleans and numerals* (and possibly other data types) as well (λNB).
- It is easy to *convert back and forth* between the two different implementations of booleans and numerals.
  - e.g., to turn a Church boolean into a primitive Boolean

    realbool = λb. b true false;

  - To go the other direction, we use an if expression:

    churchbool = λb. if b then tru else fls

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?
- Does every term evaluate to a normal form?

Divergence

$$\text{Omega} = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

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

- Terms with no normal form are said to diverge.

- Divergent computation does not seem very useful in itself. However, there are **variants** of omega that are **very useful** ...

Recursion
in the Lambda Calculus

Recursion

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

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

$$Y_f =$$

$$\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}$$

$$\longrightarrow$$

$$f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})$$

$$\longrightarrow$$

$$f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))$$

$$\longrightarrow$$

$$f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})))$$

$$\longrightarrow$$

$$\ldots$$

Recursion

- Y_f is still not very useful, since (like omega), all it does is diverge.
  - It works for the evaluation strategies like call-by-name, but fails under the call-by-value strategy. This is because the expression (λx.f (x x)) (λx.f (x x)) attempts to evaluate the argument, resulting in an infinite loop.

- Is there any way we could "slow it down" (to avoid infinite loops)?
  - We can achieve this by introducing an additional **delay wrapper** function, ensuring that the argument is evaluated only at the time of the function call.

Recursion: Delaying divergence

$$\text{delay} = \lambda y.\ \text{omega}$$

Note that delay is a *value* — it will only diverge when actually applying it to an argument, i.e., we can safely pass it as an argument to other functions, return it as a result from functions, etc.

$$(\lambda p.\ \text{fst}\ (\text{pair}\ p\ \text{fls})\ \text{tru})\ \text{delay}$$
$$\longrightarrow$$
$$\text{fst}\ (\text{pair delay fls})\ \text{tru}$$
$$\longrightarrow$$
$$\text{delay tru}$$
$$\longrightarrow$$
$$\text{omega}$$
$$\longrightarrow$$
$$\ldots\ldots$$

Recursion: Delaying divergence

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

$$\text{omegav} =$$
$$\lambda y.\left(\lambda x.\,(\lambda y.\, x\; x\; y)\right)\left(\lambda x.\,(\lambda y.\, x\; x\; y)\right) y$$

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

$$\text{omegav } v =$$
$$\underline{(\lambda y.\,(\lambda x.\,(\lambda y.\, x\; x\; y))\;\; (\lambda x.\,(\lambda y.\, x\; x\; y))\; y)\;\; v}$$
$$\longrightarrow$$
$$\underline{(\lambda x.\,(\lambda y.\, x\; x\; y))\;\; (\lambda x.\,(\lambda y.\, x\; x\; y))}\;\; v$$
$$\longrightarrow$$
$$\lambda y.\,(\lambda x.\,(\lambda y.\, x\; x\; y))\;\; (\lambda x.\,(\lambda\, y.\, x\; x\; y))\; y)\;\; v$$
$$=$$
$$\text{omegav } v$$

Recursion: another Delayed variant

- Suppose f is a function, define

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

by combining the "added f" from Y_f with the "delayed divergence" of omegav.

- apply Z_f to an argument v, something interesting happens:

$$Z_f\ v =$$

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

$$\longrightarrow$$

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

$$\longrightarrow$$

$$f\ \ (\lambda y.\ \ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ \ (\lambda x.\ f\ \ (\lambda y.\ x\ x\ y))\ y)\ v$$

$$\text{i.e.},\ \ f\ \ Z_f\ \ v$$

Recursion: another Delayed variant

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

Since Z_f and v are both *values*, the next computation step will be **the reduction of** $f\ Z_f$ — that is, f **gets to do some computation** before we “diverge”

Recursion: Generic Z

If we define

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

i.e.,

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

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

$$Z\ f \longrightarrow Z_f$$

Recursion

- Fixed-point combinator

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

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

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

Z here is essentially the same as the fix given in the textbook

As a useful generalization of omega combinator, fix can be used to help define recursive functions

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.

Recursion

$$\text{fix} = \lambda f.\ \big(\lambda x.\ f\ (\lambda y.\, x\, x\, y)\big)\ \big(\lambda x.\ f\ (\lambda y.\, x\, x\, y)\big)$$

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

- Assuming call-by-value
  - (x x) in Y_f is not a value
  - while $(\lambda y.\, x\, x\, y)$ is a value
  - Y_f will diverge for any **f**

Formalities
(Formal Definitions)

Syntax (free variables)

Substitution

Operational Semantics

Syntax

- **Definition** [Terms]:

  Let $\mathcal{V}$ be a *countable set* of variable names.

  The set of terms is *the smallest set* $\mathcal{T}$ such that

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

Syntax

- **Definition:** Free Variables of term t, written as $FV(t)$:

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

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

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

Substitution

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

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

Example:

$[x \mapsto y\ z]\ (\lambda y.\ x\ y)$

$= [x \mapsto y\ z]\ (\lambda w.\ x\ w)$

$= \lambda w.\ y\ z\ w$

Operational Semantics

Syntax

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

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-APP2)}$$

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

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

- Read through and understand Chapter 5.
- Do exercise 5.3.3 & 5.3.8 in Chapter 5.

5.3.3 EXERCISE [★★]: Give a careful proof that $|FV(\mathtt{t})| \leq size(\mathtt{t})$ for every term `t`. □

5.3.8 EXERCISE [★★]: Exercise 4.2.2 introduced a "big-step" style of evaluation for arithmetic expressions, where the basic evaluation relation is "term `t` evaluates to final result `v`." Show how to formulate the evaluation rules for lambda-terms in the big-step style. □