Principles of Programming Languages

Lecture 03

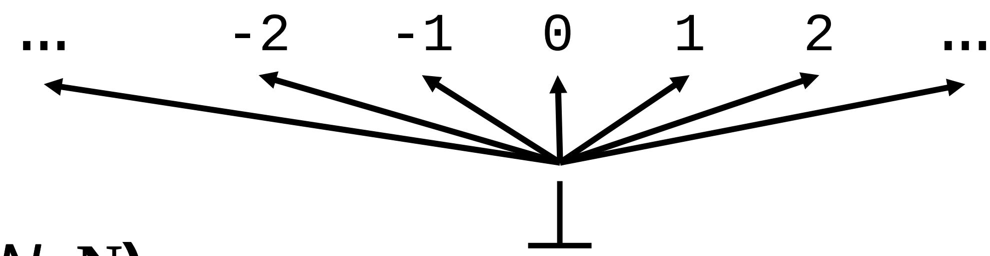
Theoretical Foundations

Domains

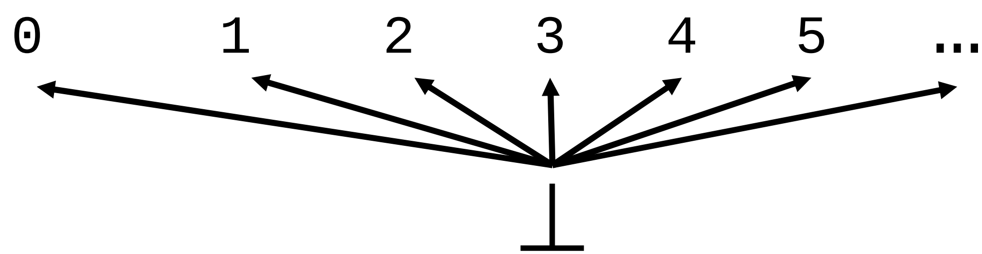
- Semantic model of a data type: *semantic domain*
  - Examples: `Integer`, `Natural`, `Truth-Value`
- Domains D are more than a set of values
  - Have an ``information ordering'' $\sqsubseteq$ on elements—a partial order--where $x \sqsubseteq y$ means (informally): ``y is as least as well-defined as x''
  - There is a ``least defined'' element $\perp$ such that $(\forall\, x)\ \perp \sqsubseteq x$ which represents an undefined value of the data type
    - Or the value of a computation that fails to terminate
    - ``bottom''
  - Domains are *complete:* every monotone (non-decreasing) sequence $x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ in D has in D a *least upper bound*, i.e., an element denoted $l = \bigcup x_i$ such that $(\forall x_i)\ x_i \sqsubseteq l$ and l is least element with this property
    - D sometimes called a ``complete partial order'' or CPO

Primitive Domains

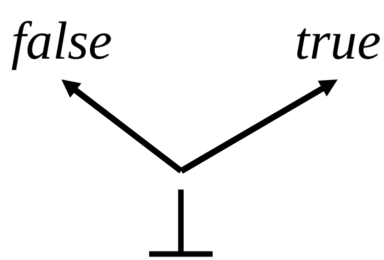
- Integer (*Z*, **Z**)



- Natural (*N*, **N**)



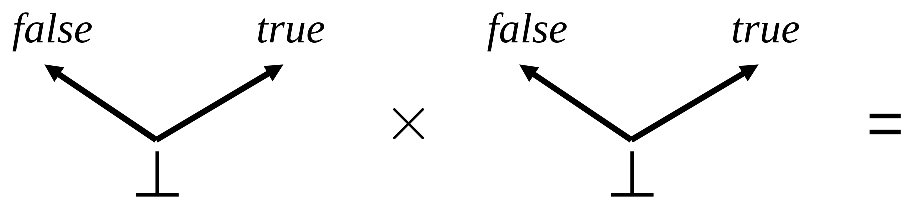
- Truth-Value (Boolean, *B*, **B**)

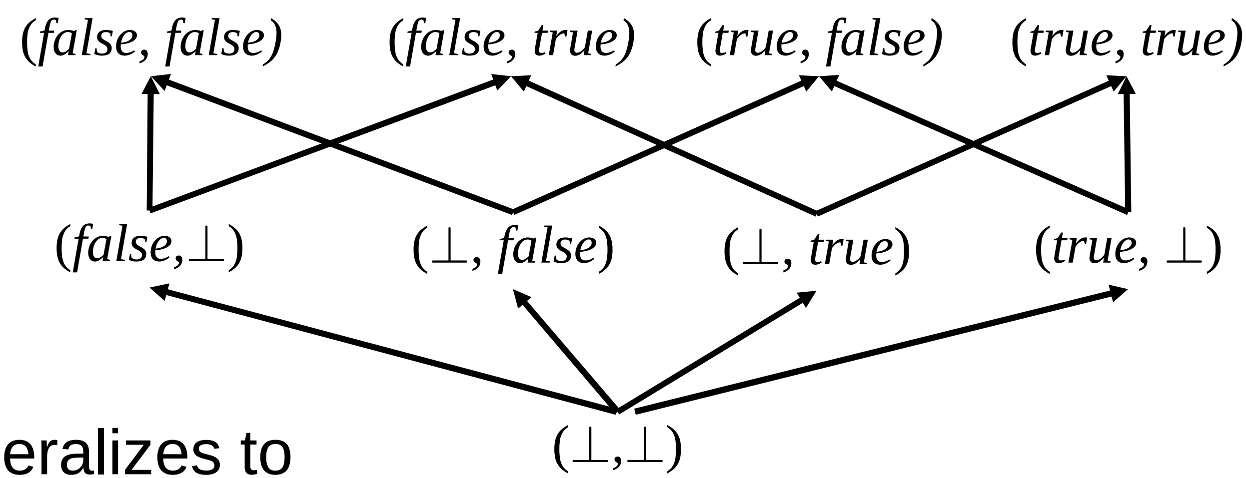


Cartesian Product Domains

- $D_1 \times D_2 = \{(x,y) \mid x \in D_1, y \in D_2\}$
  - $(x_1, y_1) \sqsubseteq (x_2, y_2) \quad \Leftrightarrow \quad x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$
  - bottom $(\bot, \bot)$

- Truth-Value $\times$ Truth-Value

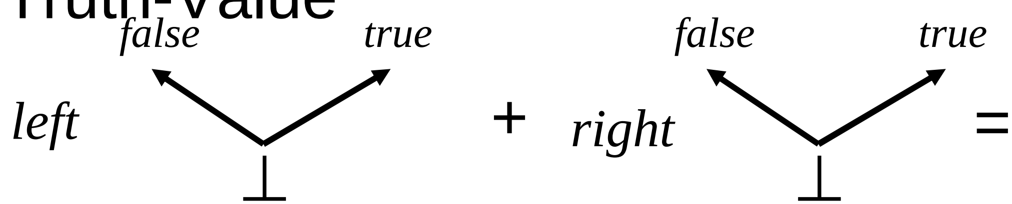


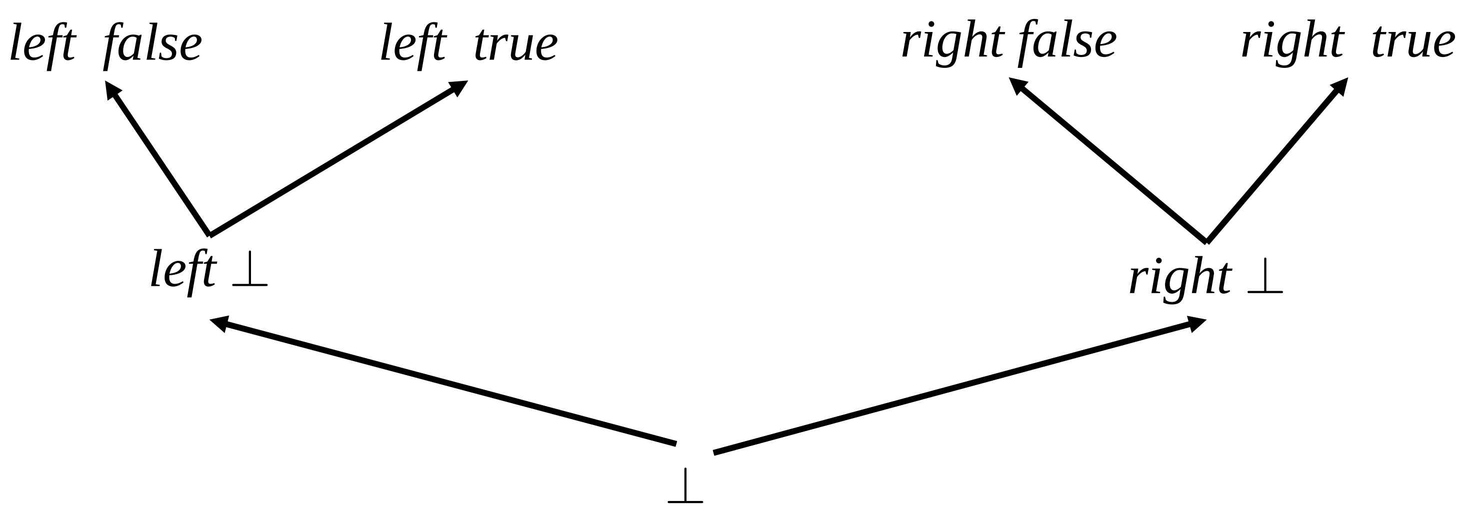


- Generalizes to
  - $D_1 \times D_2 \times D_3 \times \cdots \times D_n$

Disjoint Union (Tagged Union) Domains

- $left\ D_1 + \ right\ D_2 = \{(left, x) \mid x \in D_1\} \cup \{(right, y) \mid y \in D_2\}$
  - $left\ x_1 \sqsubseteq left\ x_2 \Leftrightarrow x_1 \sqsubseteq x_2 \qquad right\ y_1 \sqsubseteq right\ y_2 \Leftrightarrow y_1 \sqsubseteq y_2$
  - bottom $\perp$
- *left* Truth-Value **+** *right* Truth-Value





Disjoint Union Domains (cont.)

- Convention: tags are also names for *mappings* (*injections*)—tagging operators

$$D = left\ D_1 + \ right\ D_2$$

  - $$left : D_1 \rightarrow \quad D$$
    $$left(x_1) = (left, x_1) \quad [= left\ x_1]$$
    $$right : D_2 \rightarrow \quad D$$
    $$right(y_2) = (right, y_2) \quad [= right\ y_2]$$

Disjoint Union Domains (cont.)

- **Example:** union domains needed when data domains consist of different type elements (union types): imagine a very simple programming language in which the only things that can be named (denoted) are non-negative integer constants, or variables having such as values.

$$\text{Denotable} = nat\ \text{Natural} + \ var\ \text{Location}$$

  - Dereferencing operation:
    - Constant values dereference directly to their values
    - Variables (locations) dereference to their *contents* in memory

$$dereference : \text{Store} \times \text{Denotable} \ \rightarrow \ \text{Natural}$$

$$dereference(sto, nat\ val) = val \quad \text{(no dependence on memory)}$$

$$dereference(sto, var\ loc) = \ fetch(sto, loc)$$

Sequence Domains

- Consist of homogenious tuples of lengths 0, 1, … over a common domain D
  - $D^0 \triangleq \{()\} \triangleq \text{Unit}$ $\quad D^1 \triangleq D$ $\quad D^{n+1} \triangleq D \times D^n$

    $D^* \triangleq D^0 + D^1 + D^2 + \cdots$
  - Examples: $\text{Array-Value} = \text{Value}^*$

    $\text{String} = \text{Character}^*$
  - Concatenation operator

$\bullet : D^* \times D^* \to D^*$

$(d_1, d_2, \ldots, d_m) \bullet (e_1, e_2, \ldots, e_n) = (d_1, d_2, \ldots, d_m, e_1, e_2, \ldots, e_n)$

  - Empty tuple *nil* $\quad nil = ()$

    $nil \bullet x = x \bullet nil = x \qquad x \in D^*$

Functions

- A function f from *source domain* X to *target domain* Y is
  - A subset of $X \times Y$: $f \subseteq X \times Y$ (a relation)
  - Single-valued: $(\forall x, y, z)\ (x, y) \in f \wedge (x, z) \in f \Rightarrow y = z$
  - Source domain X
  - Target domain Y
  - ***Signature*** $f : X \to Y$
- Domain of f: $\mathrm{dom}\ f = \{x \in X \mid \exists y \in Y\ (x, y) \in f\}$
  - If $X = \mathrm{dom}\, f$ then f is *total* ; otherwise *partial*
- To define (specify) a function, need to give
  - Source X
  - Target Y
  - Mapping rule $x \to f(x)$

Functions (cont.)

- Example: $sq : Z \rightarrow Z$

  $sq(n) \triangleq n^2$

  - Let R = Real. *Not* the same function as:

    $sq1 : R \rightarrow R$

    $sq1(x) \triangleq x^2$

  - Let N = Natural. Not the same as:

    $\widetilde{sq} : N \rightarrow N$

    $\widetilde{sq}(i) \triangleq i^2$

```
- fun sq(n:int):int = n*n;
val sq = fn : int -> int
- sq(3);
val it = 9 : int
- sq(1.5);
stdIn:11.1-11.8 Error: operator and
operand don't agree [tycon mismatch]
operator domain: int  operand: real
in expression:  sq 1.5
- fun sq1(x:real):real = x*x;
val sq1 = fn : real -> real
-  sq1(1.5);
val it = 2.25 : real
- sq1(3);
stdIn:16.1-16.7 Error: operator and
operand don't agree [literal]
operator domain: real operand:int
in expression: sq1 3
-
```

Defining Functions

- Two views of ``mapping''
  - *Extension*: view as a collection of facts

$$square : Z \to Z$$

$$square = \{(0,0),(1,1),(-1,1)(2,4),(-2,4)(3,9),\ldots\} \subseteq N \times N$$

  - *Comprehension*: view as a rule of mapping

$$square : Z \to Z$$

    - *Combinator form:* $square(x) \triangleq x * x$
    - *λ-abstraction form:* $square \triangleq \lambda y. y * y$
  - Typed λ-calculus

$$square : Z \to Z \triangleq \lambda y : Z. y * y$$

  - In ML
    - λ = `fn`
    - `.` = `=>`
    - $(\lambda x \bullet \ e)$ = `(fn x => e)`

```
- fun square(x) = x*x;
val square = fn : int -> int
- square(4);
val it = 16 : int
- val square = (fn y => y*y);
val square = fn : int -> int
- square(5);
val it = 25 : int
```

Defining Functions (cont.)

- Examples of λ-notation defining functions

$$dist : R \times R \to R \triangleq \lambda(u,v).\sqrt{(u-v)^2}$$

$$double : Z \to Z \triangleq \lambda n : Z. n + n$$

$$successor : Z \to Z \triangleq \lambda n : Z. n + 1$$

$$twice : (Z \to Z) \to (Z \to Z) \triangleq \lambda f : (Z \to Z). \lambda n : Z. f(f(n))$$

$$compose : (Z \to Z) \to ((Z \to Z) \to (Z \to Z)) \triangleq \lambda f.(\lambda g.(\lambda n. f(g(n))))$$

- $\therefore$
  - $dist(0, \,) : R \to R \triangleq \lambda v.\sqrt{v^2}$
  - $twice(double) : Z \to Z \triangleq \lambda n : Z. 4 \cdot n$
  - $twice(successor) : Z \to Z \triangleq \lambda n : Z. n + 2$
  - $(compose(successor))(double) : Z \to Z \triangleq \lambda n : Z. 2 \cdot n + 1$

Defining Functions (cont.)

```
- val double = (fn n => n+n);
val double = fn : int -> int
- val successor = (fn n => n+1);
val successor = fn : int -> int
- val twice = (fn f => (fn n => f(f(n))));
val twice = fn : ('a -> 'a) -> 'a -> 'a
- val twice = (fn f : int -> int => (fn n : int => f(f(n))));
val twice = fn : (int -> int) -> int -> int
- val td = twice(double);
val td = fn : int -> int
- td(3);
val it = 12 : int
- val ts = twice(successor);
val ts = fn : int -> int
- ts(3);
val it = 5 : int
```

Defining Functions (cont.)

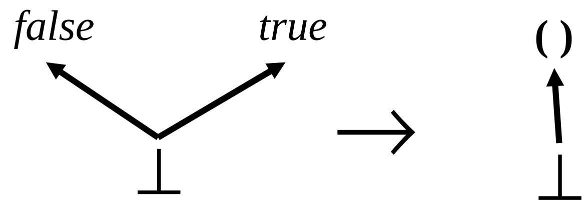
```
- fun double(n) = n+n;
val double = fn : int -> int
- fun successor(n) = n+1;
val successor = fn : int -> int
- fun twice(f)(n) = f(f(n));
val twice = fn : ('a -> 'a) -> 'a -> 'a
- fun twice(f : int -> int )(n : int) = f(f(n));
val twice = fn : (int -> int) -> int -> int
- fun td = twice(double);
stdIn:39.5-39.7 Error: can't find function arguments in clause
- twice(double)(3);
val it = 12 : int
- twice(successor)(3);
val it = 5 : int
- fun compose(f)(g)(n) = f(g(n));
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- val csd = compose(successor)(double);
val csd = fn : int -> int
- csd(3);
val it = 7 : int
```

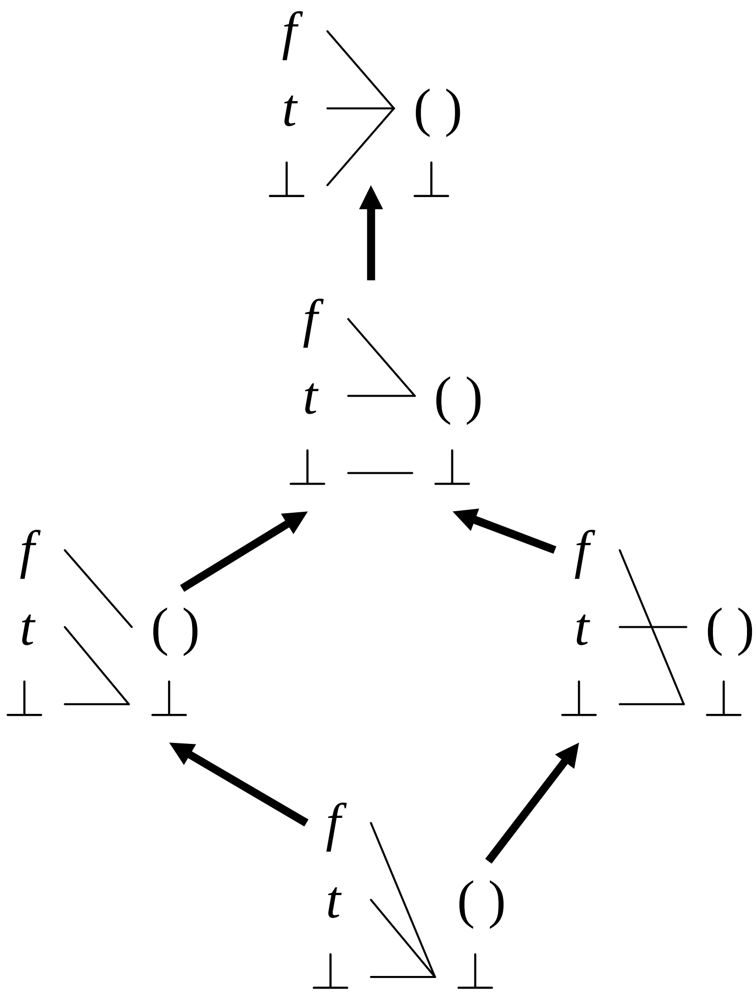
Function Domains

- Elements are functions $f : D_1 \to D_2$ from a domain to another
- Domain of functions denoted $D_1 \to D_2$ (or $[D_1 \to D_2]$)
- Not *all* functions—must be *monotone:* if $f \in D_1 \to D_2$
  $\forall x, y \in D_1 \; x \sqsubseteq_{D_1} y \Rightarrow f(x) \sqsubseteq_{D_2} f(y)$
  - Idea: more info about argument $\Rightarrow$ more info about value
- Ordering on domain $D_1 \to D_2$
  $f \sqsubseteq_{D_1 \to D_2} g \quad \Leftrightarrow \quad (\forall x \in D_1 \; f(x) \sqsubseteq_{D_2} g(x)$
- Bottom element of domain $D_1 \to D_2$
  $$\bot_{D_1 \to D_2} \triangleq \lambda x. \bot_{D_2}$$
  - “totally undefined’’ function
- Technical requirement: functions must be *continuous*:
  $\forall$ monotone chains $x_1 \sqsubseteq x_2 \sqsubseteq \cdots \Rightarrow f(\bigcup x_i) = \bigcup f(x_i)$

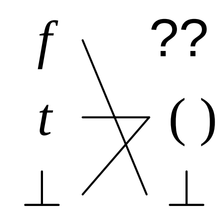
Function Domains (cont.)

- (Truth-Value $\rightarrow$ Unit)



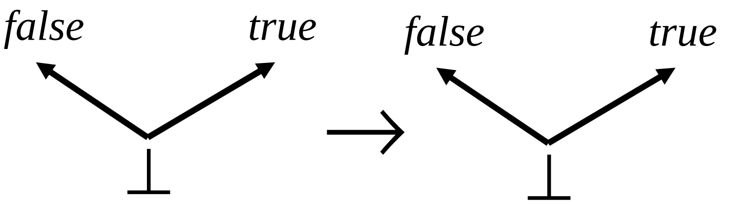


Models procedures (void functions) that take a boolean and return, or not …



Function Domains (cont.)

- (Truth-Value $\rightarrow$ Truth-Value)



Exercise!

Be careful about monotonicity!

λ-Calculus

- Let e be an expression containing zero or more occurrences of variables $x_1, x_2, \ldots, x_n$
- Let $e[a / x]$ denote the result of replacing each occurrence of x by a
- The *lambda expression* $\lambda x_1 x_2 \ldots x_n . e$
  - Denotes (names) a function
  - Value of the λ-expression *is* a function f such that

$$\forall a_1 a_2 \ldots a_n \quad f(a_1, a_2, \ldots, a_n) =$$

$$\text{value of expression } e[a_1 / x_1, a_2 / x_2, \ldots, a_n / x_n]$$

  - Provides a direct description of a function object, not indirectly via its behavior when applied to arguments

λ-Calculus Examples

- Example:

$$sq = \text{``that function that if applied to } a \text{ results in } a \cdot a\text{''}$$

$$sq = \lambda x.(x \cdot x)$$

- Example:

$$zval = \text{``the functional that when applied to function } f \text{ results in } f(0)\text{''}$$

$$zval = \lambda f.\, f(0)$$

- Example:

$$compose = \text{``the functional that when applied to functions } f \text{ and } g \text{ results in the funtion } \lambda x. f(g(x))\text{''}$$

$$compose = \lambda f.(\lambda g.(\lambda x.(f(g(x)))))$$

λ-Calculus Examples (cont.)

- *Functionals*: functions that take other functions as arguments or that return functions; also *higher-type functions*
- Example: the definite integral $\int_0^1 f(u)du \quad (f : R \to R)$

$$\int_0^1 : (R \to R) \to R$$

$$\int_0^1 = \lambda f . \int_0^1 f(u)du$$

- Example: the indefinite integral $\int_0^x f(u)du \quad (f : R \to R)$

$$\int_0^x u^2 du = \frac{x^3}{3} \qquad \int_0 : (R \to R) \to (R \to R)$$

$$\int_0 (\lambda z . z^2) = \lambda x . \frac{x^3}{3} \qquad \int_0 = \lambda f . \left(\lambda x . \int_0^x f(u)du \right)$$

λ-Calculus Examples (cont.)

- Example: integration routine

$$trapezoidal : R \times R \times R \times (R \to R) \to R$$

$$trapezoidal = \lambda habf.h \cdot \left(\sum_{i=0}^{[(b-a)/n]} f(a + i \cdot h) - (f(a) + f(b))/2 \right)$$

- Example: summation "operator" $\sum_{i=0}^{n} f(i) \quad (f : N \to R)$

$$\sum_0^n : (N \to R) \to R$$

$$\sum_0^n (\lambda m.1) = n + 1 \qquad \sum_0^n (\lambda m.m) = \frac{n(n+1)}{2}$$

$$\sum_0 : (N \to R) \to (N \to R)$$

$$\sum_0 (\lambda m.1) = \lambda n.(n+1)$$

λ-Calculus Examples (cont.)

- Example: derivative operator

  ✗ $Dx^n = nx^{n-1} \quad D3^2 = 0\ (!)$

  $$D : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

  $$D\left(\lambda x.x^n\right) = \left(\lambda x.nx^{n-1}\right)$$

- Example: indexing "operator"

  Postfix `[i]`: `IntArray` → `Integer`

  `[i]` = $\lambda a.a$`[i]`

- Example: "funcall" or "apply" operator
  - Signature $apply : (A \to B) \to A \to B$
  - Combinator definition $apply(f)(a) = f(a)$
  - Lambda definition $apply = \lambda f.\lambda a.f(a)$

Currying (Curry 1945;Schönfinkel 1924)

- Transformation reducing a multi-argument function to a cascade of 1-argument functions

$$curry : (X \times Y \rightarrow Z) \rightarrow (X \rightarrow (Y \rightarrow Z))$$

$$f : X \times Y \rightarrow Z \qquad curry(f) : (X \rightarrow Y) \rightarrow Z$$

$$f = \lambda(x,y).f(x,y) \qquad curry(f) = \lambda x.\lambda y.f(x,y)$$

$$\therefore \quad curry = \lambda f.\lambda x.\lambda y.f(x,y)$$

- Examples

$$curry("-") = \lambda x.\lambda y.(x-y) \quad ["-" = \lambda(x,y)(x-y)]$$

$$curry("-")(3) = \lambda y.(3-y)$$

$$curry("*")(2) = \lambda z.(2 \cdot z) = double$$

- Applying $curry(f)$ to first argument a yields a *partially evaluated* function *f* of its second argument: *f(a,-)*

$$\lambda y.f(a,y)$$

Currying (cont.)

- Example: machine language curries

$$+ = \lambda a.\lambda b.(a+b)$$

  - Assume *contents*(A)=*a* and *contents*(B)=*b*

$$\texttt{r0} = \lambda a.\lambda b.(a+b)\begin{cases}\texttt{LDA A} \\ \texttt{ADD B}\ \}\ \texttt{r0} = \lambda b.(a+b)\end{cases}$$

λ-Calculus Syntax

- `Lambda`=*expression* `Id`=*identifier* Ab=*abstraction* Ap=*application*

```
Lambda →Id | Ab | Ap
Ab → (λ Id . Lambda)
Ap → (Lambda Lambda)
Id →  x | y | z | …
```

- Examples $((xy)z)$

$$(fx)$$

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

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

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

$$(\lambda x.(\lambda y.(sqrt((plus(square\ x))(square\ y)))))$$

λ-Calculus: simplified notation

- Conventions for dropping () $\Rightarrow$ disambiguation needed
- Rules for dropping parentheses
  - Application groups L to R

$$xyz \triangleq ((xy)z) \qquad x(yz) \triangleq (x(yz))$$

  - Application has higher precedence than abstraction

$$\lambda x.yz \triangleq (\lambda x.(yz)) \qquad (\lambda x.y)z \triangleq ((\lambda x.y)z)$$

  - Abstraction groupts R to L

$$\lambda x.\lambda y.\lambda z.e \triangleq (\lambda x.(\lambda y.(\lambda z.e)))$$

  - Consecutive abstractions collapse to single λ

$$\lambda xyz.e \triangleq \lambda x.\lambda y.\lambda z.e$$

  - Example:

$$\mathbf{S} \triangleq (\lambda x.(\lambda y.(\lambda z.((xz)(yz))))) \quad \Rightarrow$$

$$\mathbf{S} = \lambda xyz.(xz)(yz) = \lambda xyz.xz(yz)$$

$$\neq \lambda xyz.(xz)yz = (\lambda x.(\lambda y.(\lambda z.(((xz)y)z))))$$

Variables: Free, Bound, Scope

- Metavariables: I: `Id` L: `Lambda`
- Type the semantic function

  occurs: `Lambda`→(`Id`→*B*)
- one rule per syntactic case (syntax-driven)
  - ⟦ ⟧ are traditional around *syntax* argument

$$occurs[\![I]\!]\mathsf{x} = (I = \mathsf{x})$$

$$occurs[\![\lambda I.L]\!]\mathsf{x} = occurs[\![L]\!]\mathsf{x}$$

$$occurs[\![L_1\ L_2]\!]\mathsf{x} = occurs[\![L_1]\!]\mathsf{x} \ \vee \ occurs[\![L_2]\!]\mathsf{x}$$

- *Note:* In expession $\lambda x.y$ the variable x does *not* occur!

Variables: Free, Bound, Scope (cont.)

- *occurs_bound* : `Lambda`$\rightarrow$(`Id`$\rightarrow$*B*)

$$occurs\_bound[\![I]\!]\mathtt{x} = false$$

$$occurs\_bound[\![\lambda I.L]\!]\mathtt{x} = (\mathtt{x} = I) \wedge occurs\_free[\![L]\!]\mathtt{x}$$

$$occurs\_bound[\![L_1\ L_2]\!]\mathtt{x} = occurs\_bound[\![L_1]\!]\mathtt{x} \vee occurs\_bound[\![L_2]\!]\mathtt{x}$$

- *occurs_free* : `Lambda`$\rightarrow$(`Id`$\rightarrow$*B*)

$$occurs\_free[\![I]\!]\mathtt{x} = (I = \mathtt{x})$$

$$occurs\_free[\![\lambda I.L]\!]\mathtt{x} = (\mathtt{x} \neq I) \wedge occurs\_free[\![L]\!]\mathtt{x}$$

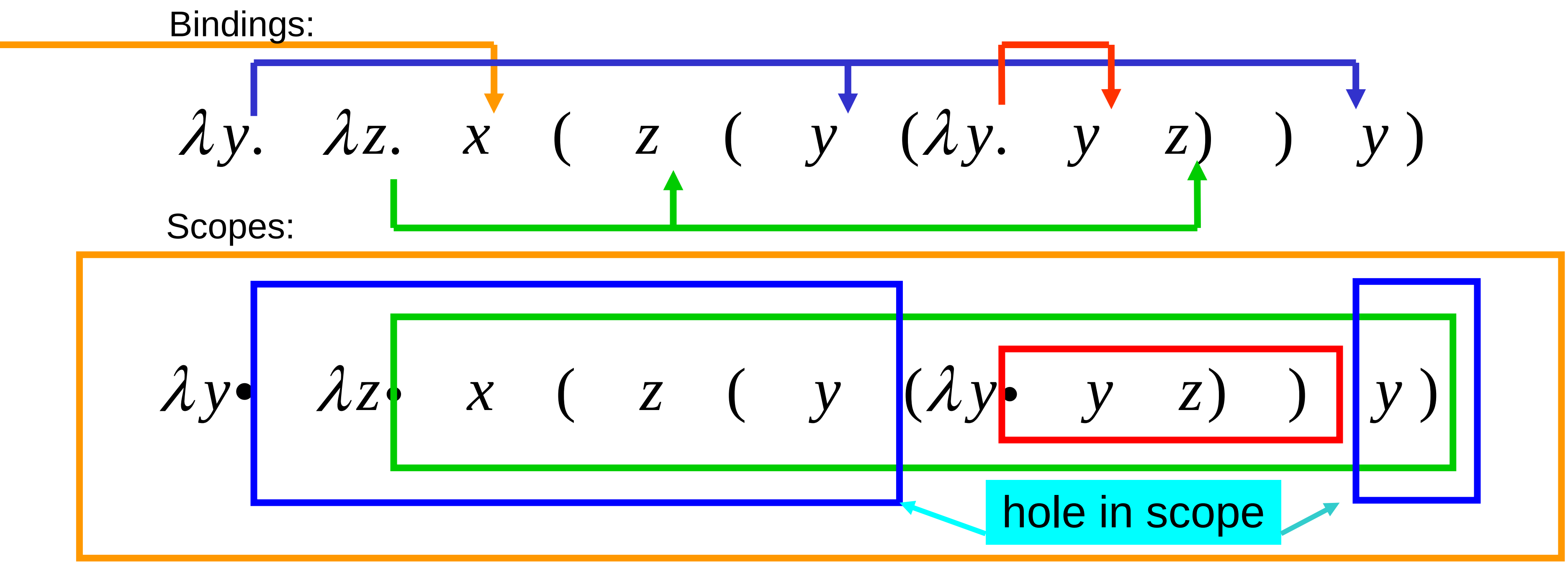
$$occurs\_free[\![L_1\ L_2]\!]\mathtt{x} = occurs\_free[\![L_1]\!]\mathtt{x} \vee occurs\_free[\![L_2]\!]\mathtt{x}$$

Variables: Free, Bound, Scope (cont.)

- The notions *free* x, *bound* x, *occur* x are *relative to a given expression or subexpression*
- Examples:
  - $(\lambda u.(\lambda x.ux)xu)$
  - $(\lambda x.y)$
  - $(\lambda x.(\lambda y.y))$

Variables: Scope

- The *scope* of a variable x bound by an λx prefix in an abstraction $(\lambda x\,.e)$ consists of all of e *excluding* any abstraction subexpressions with prefix λx
  - these subexpressions are known as “holes in the scope”
  - Any occurrence of x in scope is said to be *bound* by the λx prefix
- *Example*



λ-calculus: formal computation reduction rules

- α-conversion: formal parameter names do not affect meaning $\lambda x.e \underset{\alpha}{\longleftrightarrow} \lambda y.e[y/x]$

  substitute y for *free* occurrences of x in e (rename of clash occurs)

  - Ex: $\lambda x.\lambda z.a(xz) \underset{\alpha}{\longleftrightarrow} \lambda x.\lambda z'.a(xz')$

    $\underset{\alpha}{\longleftrightarrow} \lambda z.\lambda z'.a(zz') \underset{\alpha}{\longleftrightarrow} \lambda z.\lambda x.a(zx)$

- β-reduction: models application of a function abstraction to its argument $(\lambda x.e)a \underset{\beta}{\longrightarrow} e[a/x]$

  redex contractum

  - Ex: $(\lambda x.\lambda y.x)a((\lambda x.xx)(\lambda x.xx)) \underset{\beta}{\longrightarrow}$

    $(\lambda y.a)((\lambda x.xx)(\lambda x.xx)) \underset{\beta}{\longrightarrow} a$

λ-calculus: formal computation (cont.)

- η-reduction: models "extensionality"

$$\lambda x.ex \xrightarrow[\eta]{} e$$

x *not* free in e (no other x occurrences in scope)

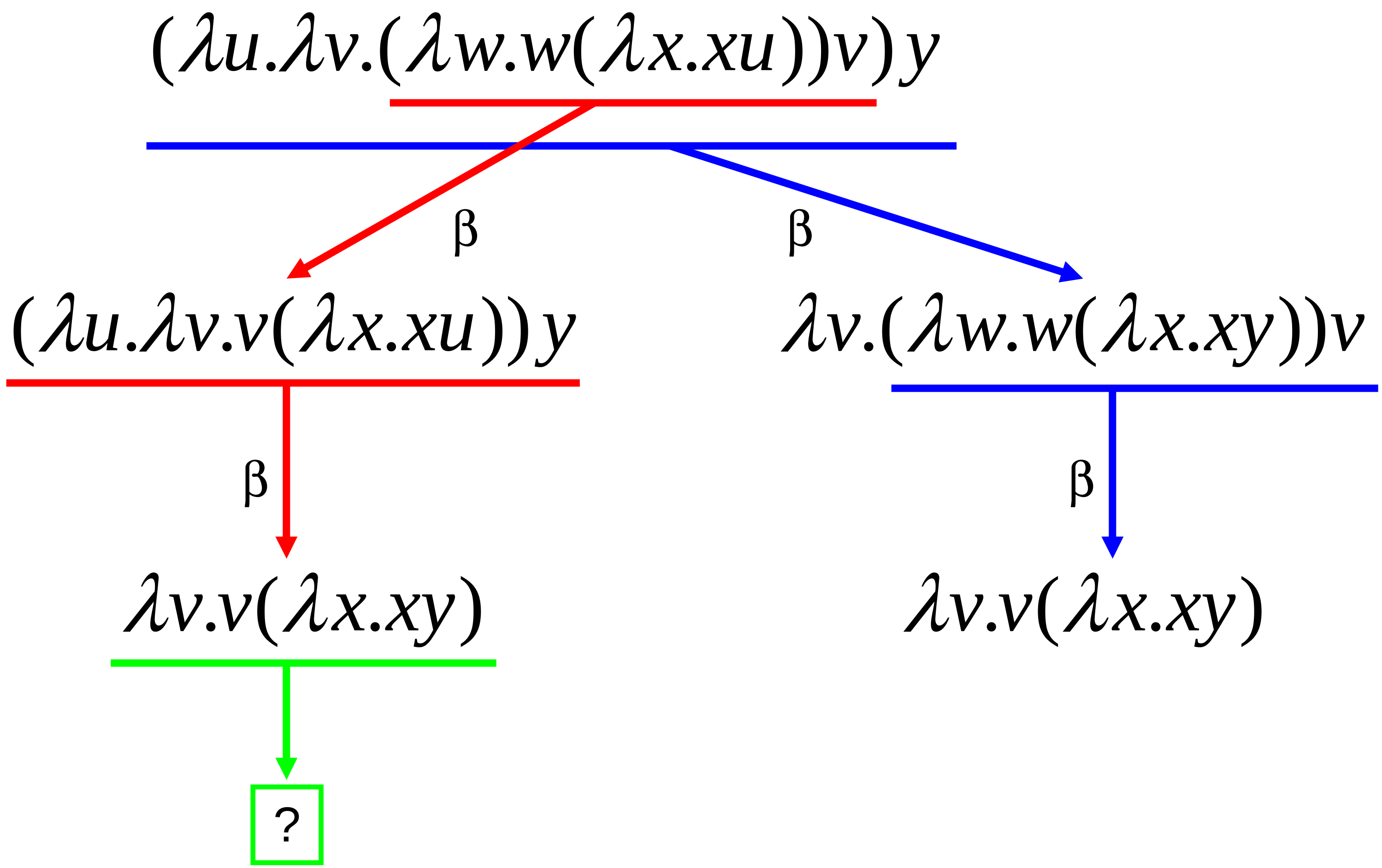
  - Ex:

$$\lambda v.(\lambda x.x(\lambda x.ux))v$$

$$\lambda v.(\lambda x.x(\lambda x.ux))v \xrightarrow{\eta} \lambda v.(\lambda x.xu)v \xrightarrow{\eta} (\lambda x.xu)$$

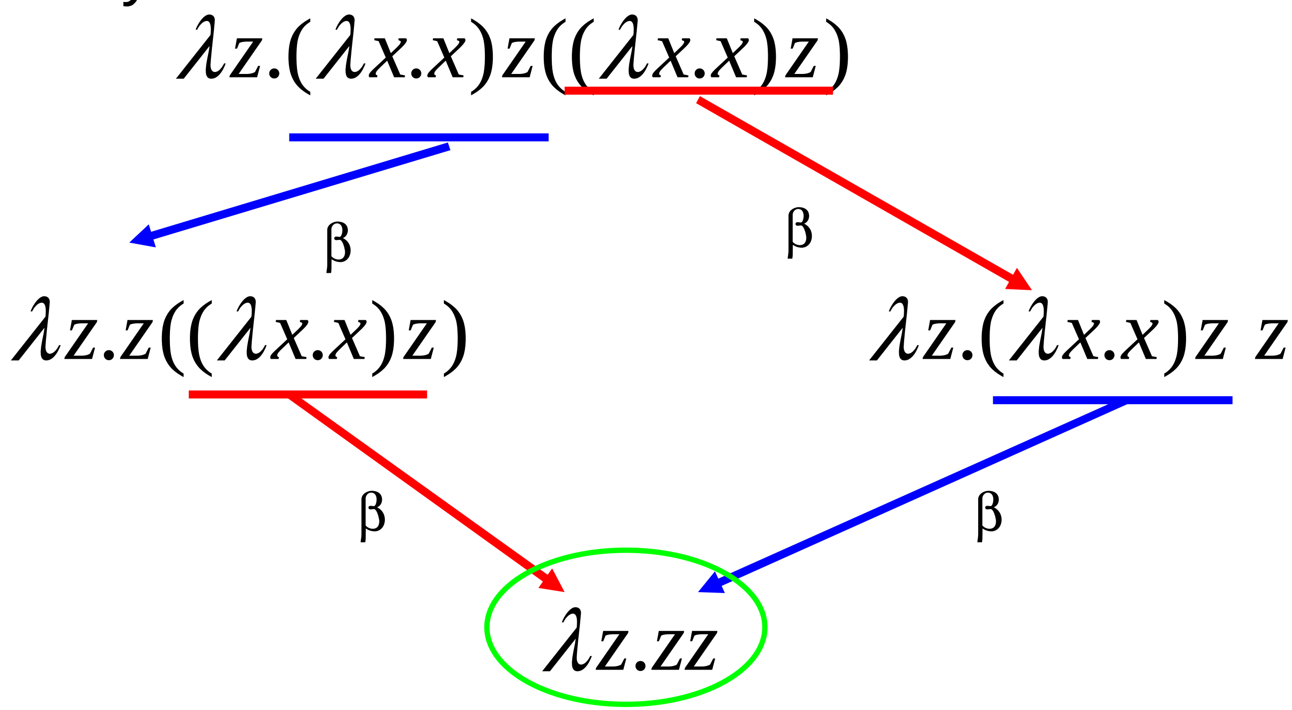
$$\lambda v.(\lambda x.x(\lambda x.ux))v \xrightarrow{\eta} (\lambda x.x(\lambda x.ux)) \xrightarrow{\eta} (\lambda x.xu)$$

Reduction is locally non-deterministic



Normal Forms

- Identify α-equivalent expressions
- Reduction: $\longrightarrow \triangleq (\underset{\beta}{\longrightarrow} \cup \underset{\eta}{\longrightarrow})$
- Reduction "as far as possible": $\overset{*}{\longrightarrow} \triangleq (\longrightarrow)^{*}$
- *Defn: Normal Form:* an expression that cannot be further reduced by $\longrightarrow$



Normal Forms Don't Always Exist

- Parallel of "non-terminating computation"
- Example: "self-application combinator": SELF $\triangleq \lambda x.xx$
- What happens with $(\text{SELF SELF}) = ((\lambda x.xx)(\lambda x.xx))$?

$$((\lambda x.xx)(\lambda z.zz))$$

$$\downarrow \beta$$

$$(\lambda z.zz)(\lambda z.zz)$$

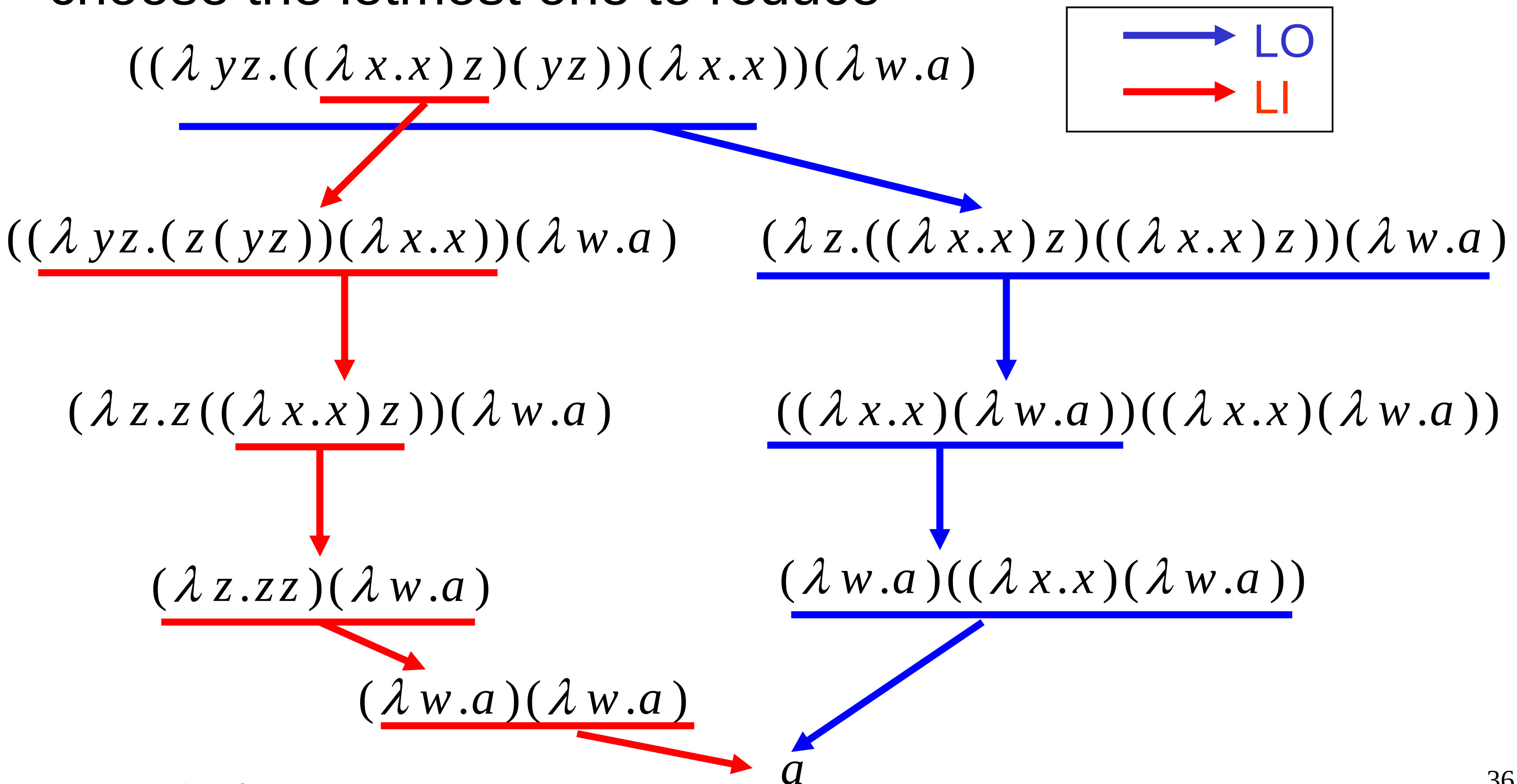
$$\downarrow \beta$$

$$\vdots$$

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

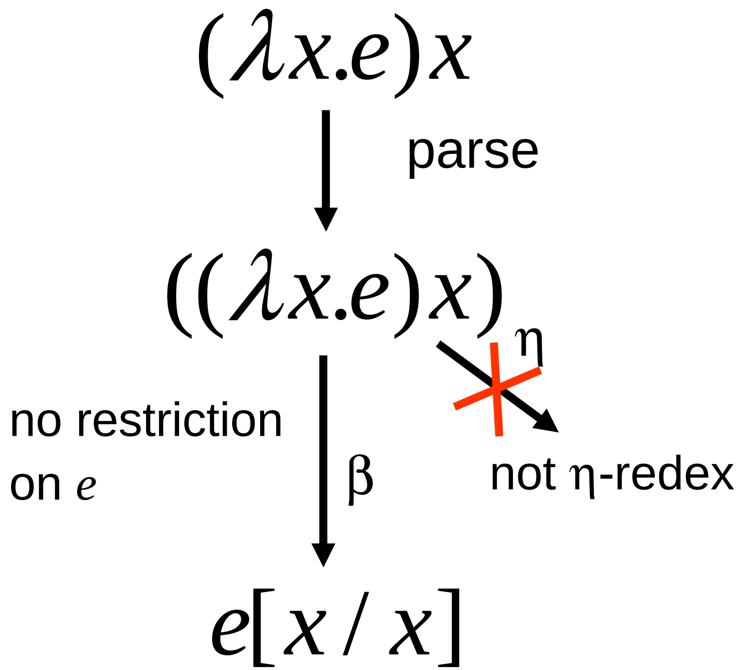
Computation Rules

- *Leftmost Innermost (LI):* among the innermost nested redexes, choose the leftmost one to reduce
- *Leftmost Outermost(LO):* among the outermost redexes, choose the letmost one to reduce



β vs η --very different

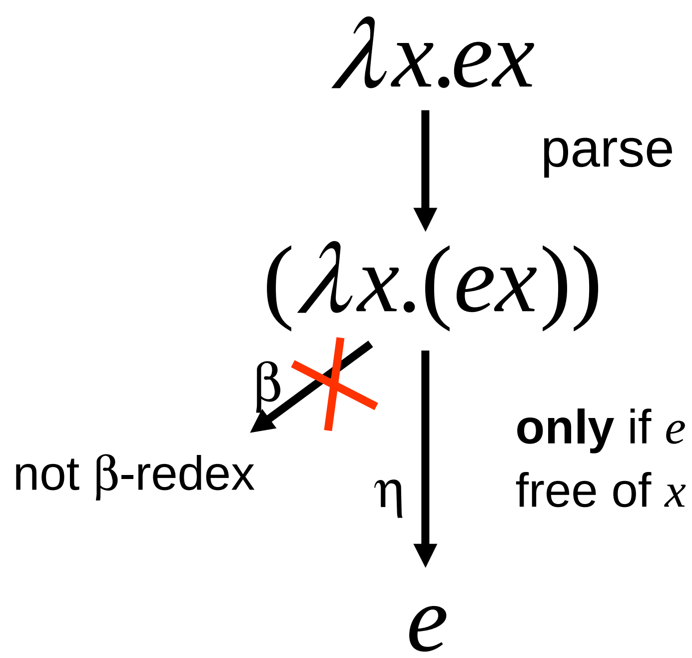
- β-reduction: abstract *e*, then apply



- Example:

```
((lambda (x) 1+) x) → 1+
```

- η-reduction: apply *e*, then abstract



- Example:

```
(lambda (x) (1+ x)) → 1+
```

Mixed β and η reductions

$$\lambda v. \quad (\lambda x. \quad x\,(\lambda x.ux)\ ) \quad v$$

↓ parse

$$(\lambda v.((\lambda x.(x\ (\lambda x.ux))\ )\quad v)\ )$$

η ↙ ↘ β

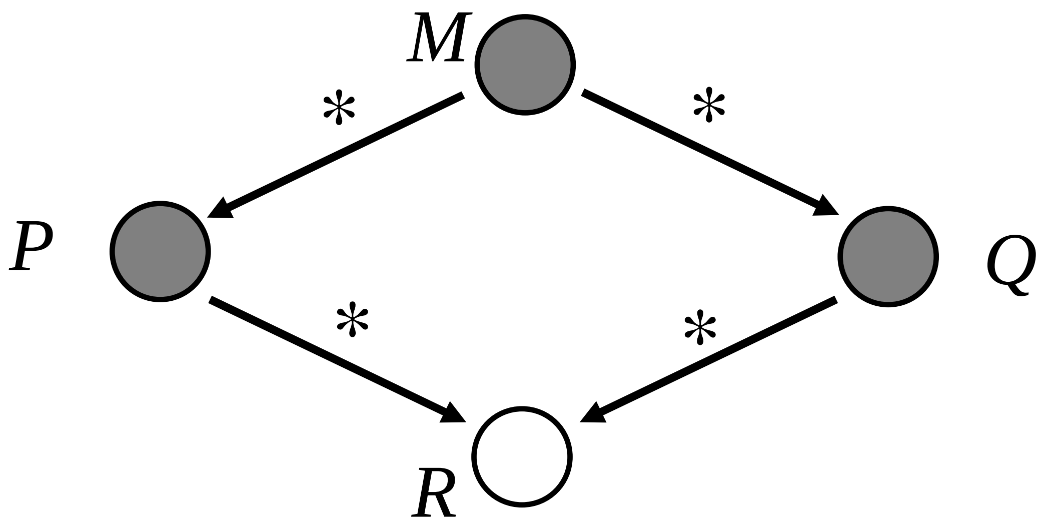
$$(\lambda v.((\lambda x.(x\ \ u)\ \ )\quad v)\ ) \qquad (\lambda v.(v\ (\lambda x.(ux))\ )\ )$$

β ↘ ↙ η

$$(\lambda v.(vu))$$

Church-Rosser: ultimate determinism

- Church-Rosser Theorem



$$\forall M, P, Q \quad M \xrightarrow{*} P \;\&\; M \xrightarrow{*} Q$$
$$\Rightarrow \exists R \quad P \xrightarrow{*} R \;\&\; Q \xrightarrow{*} R$$

- Corollary: *If* a normal form exists for an expression, it is unique (up to α-equivalence)

Church-Rosser: says ∃ reduction, not *forced*

$(\lambda x.a)(\text{SELF SELF}) = (\lambda x.a)((\lambda x.xx)(\lambda x.xx))$

LO → a LI → $(\lambda x.a)(\text{SELF SELF})$

LO → a LI → $(\lambda x.a)(\text{SELF SELF})$

LO → a LI → $(\lambda x.a)(\text{SELF SELF})$

LO → a LI → . . .

Correct computation rule: if a normal form ∃, it will be computed—ex: LO

Incorrect (but efficient):—ex: LI

LO = normal rule = CBN = lazy

Prog Lang: Haskell

LI = applicative rule = CBV=eager

Prog Lang: Scheme

Correct Computation Rule

- A rule is *correct* if it calculates a normal form, provided one exists
- **Theorem:** LO (normal rule) is a correct computation rule.
  - LI (applicative rule) is *not correct*—gives the same result as normal rule *whenever it converges to a normal form* (guaranteed by the Church-Rosser property)
  - LI is simple, efficient and natural to implement: always evaluate all arguments before evaluating the function
  - LO is very inefficient: performs lots of copying of unevaluated expressions

Typed λ-calculus

- More restrictive reduction. No $\text{SELF} = (\lambda x.xx)$

```
Type ::= (Type →Type) | PrimType
Lambda ::= Id | (Lambda Lambda)
              | (λ Id : Type . Lambda)
Id ::=  x | y | z | …
PrimType ::= int | …
```

- Type Deduction Rules

$$\frac{}{\triangleright x : \tau} \text{ (axiom)}$$

$$\frac{\triangleright M : \sigma \to \tau \quad \triangleright N : \sigma}{\triangleright (M\ N) : \tau} \text{ (appl)}$$

$$\frac{\triangleright M : \tau \quad \triangleright x : \sigma}{\triangleright (\lambda x : \sigma . M) : \sigma \to \tau} \text{ (abst)}$$

- E is *type correct* iff ∃ a type α such that $\triangleright E : \alpha$
- *Thm:* Every type correct expression has a normal form

Typed λ-calculus (cont.)

- Example: $\mathsf{SELF} = (\lambda x.xx)$ is not type correct
  - Work backward using rules. Assume $\triangleright \lambda x:\alpha.xx:\alpha \to \beta \;\&\; \triangleright x:\gamma$

$$\frac{\triangleright (xx):\beta \qquad \triangleright x:\alpha}{\triangleright \lambda x:\alpha.(xx):\alpha \to \beta} \quad \Rightarrow \quad \gamma = \alpha \Rightarrow \triangleright x:\alpha \,\&\, (xx):\beta$$

$$\frac{\triangleright x:\delta \to \beta \qquad \triangleright x:\delta}{\triangleright (xx):\beta}$$

$$\therefore \;\; \triangleright x:\delta \;\; \& \triangleright x:\alpha \Rightarrow \delta = \alpha$$

$$\therefore \triangleright x:\delta \to \beta \;\; \& \quad \triangleright \delta = \alpha \Rightarrow \triangleright x:\alpha \to \beta$$

$$\therefore \triangleright x:\alpha \,\&\, \triangleright x:\alpha \to \beta$$

  - So $\alpha = (\alpha \to \beta)$ a contradiction. Therefore there is no consistent type assignment to $\mathsf{SELF} = (\lambda x.xx)$

Typed λ-calculus (cont.)

- Example: type the expression $\lambda f.\lambda x.fx$
  - Work forward

$$\frac{\triangleright\ f:\alpha\rightarrow\beta \qquad \triangleright\ x:\alpha}{\triangleright\ (fx):\beta}\ \text{(appl)}$$

$$\frac{\triangleright\ (fx):\beta \qquad \triangleright\ x:\alpha}{\triangleright\ \lambda x:\alpha.(fx):\alpha\rightarrow\beta}\ \text{(abst)}$$

$$\frac{\triangleright\ \lambda x:\alpha.(fx):\alpha\rightarrow\beta \qquad \triangleright\ f:\alpha\rightarrow\beta}{\triangleright\ (\lambda f:\alpha\rightarrow\beta.(\lambda x:\alpha.(fx))):(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta)}\ \text{(abst)}$$

λ-Calculus: History

- Frege 1893: unary functions suffice for a theory
- Schönfinkel 1924:
  - Introduced "currying"
  - Combinators KAB $\triangleq$ A   SABC $\triangleq$ AC(BC)

    K $\triangleq \lambda ab.a$ S $\triangleq \lambda abc.ac(bc)$
  - Combinatory (Weak) Completeness: all closed λ-expressions can be defined in terms of **K,S** using application only: Let M be built up from λ, **K,S** with only x left free. Then ∃ an expression F built from **K,S** only such that $Fx=M$
- Curry 1930:
  - introduced axiom of extensionality: $\forall X \quad FX = GX \Rightarrow F = G$
  - Weak Consistency: **K**=**S** is not provable

λ-Calculus: History (cont.)

- Combinatory Completeness Example:
  - Composition functional: $\mathsf{BXYZ} \triangleq \mathsf{X(YZ)}$ $\qquad \mathsf{B} \triangleq \lambda fgx.f(gx)$
  - Thm: $\mathsf{B} = \mathsf{S(KS)K}$
  - Prf: Note that $\mathsf{K} \triangleq \lambda xy.x \Rightarrow \mathsf{KS} = (\lambda xy.x)\mathsf{S} = \lambda y.\mathsf{S}$

So $\mathsf{S(KS)} = (\lambda xyz.xz(yz))(\mathsf{KS}) = (\lambda yz.((\mathsf{KS})z)(yz))$

$$= (\lambda yz.((\lambda y.\mathsf{S})z))(yz) = \lambda yz.\mathsf{S}(yz)$$

and so $\mathsf{S(KS)K} = (\lambda yz.\mathsf{S}(yz))\mathsf{K} = \lambda z.\mathsf{S}(\mathsf{K}z) = \lambda z.\mathsf{S}(\lambda y.z)$

$$\stackrel{\alpha}{=} \lambda v.\mathsf{S}(\lambda w.v) = \lambda v.(\lambda xyz.xz(yz))(\lambda w.v)$$

$$= \lambda v.(\lambda yz.(\lambda w.v)z(yz)) = \lambda v.(\lambda yz.v(yz))$$

$$= \lambda v.\lambda y.\lambda z.v(yz) \stackrel{\alpha}{=} \lambda f.\lambda g.\lambda x.f(gx) \qquad \square$$

- Exercise: define $\mathsf{C} \triangleq \lambda xyx.xzy$ & show $\mathsf{C} = \mathsf{S(BBS)(KK)}$

λ-Calculus: History (cont.)

- Church 1932:
  - If $Fx=M$ call F "$\lambda x.M$"
  - Fixed Point Theorem: Given F can construct a Φ such that $\Phi=F\Phi$
  - Discovered fixed point (or "paradoxical") combinator:

$$\mathrm{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

  - Exercise: Define $\Phi=\mathrm{Y}F$ and show by reduction that $\Phi=F\Phi$
- Church & Rosser 1936: *strong* consistency ≡ Church-Rosser Property.
  - Implies that α-equivalence classes of normal forms are disjoint
  - ∴ provides a "syntactic model" of computation
- Martin-Löf 1972: simplified C-R Theorem's proof

λ-Calculus: History (cont.)

- Church 1932: Completeness of the λ-Calculus
  - *N* can be embedded in the λ-Calculus

$$n \triangleq \lambda f.\lambda x.\underbrace{f(f(\ldots(f}_{n}\ x)\ldots))$$

  - Church 1932/Kleene 1935: recursive definition possible:

$$\left.\begin{aligned} F(1) &= GA \\ F(n+1) &= G(Fn) \end{aligned}\right\}\text{K-C-G ``general recursion"}$$

$$Thm:\ F \text{ is ``}\lambda\text{-definable":}\quad F = \lambda x.xGA \quad (!)$$

  - Check:

$$F1 = (\lambda x.xGA)1 \to 1GA \to (\lambda f.\lambda x.fx)GA$$

$$\to (\lambda x.Gx)A \to GA$$

$$F2 \to 2GA \to (\lambda f.\lambda x.f(fx))GA$$

$$\to (\lambda x.G(Gx))A \to G(GA)$$

λ-Calculus: History (cont.)

- Church-Rosser Thm (1936) ⇒ λ-Calculus functions are ``computable’’ (aka ``recursive”)
- *Church’s* *<u>Thesis</u>:* The ``effectively computable” functions from *N* to *N* are exactly the “λ-Calculus definable” functions
  - a strong assertion about “completeness”; all programming languages since are “complete” in this sense
- Evidence accumulated
  - Kleene 1936: general recursive ⇔ λ-Calculus definable
  - Turing 1937: λ-Calculus definable ⇔Turing-computable
  - Post 1943, Markov 1951, etc.: many more confirmations
  - Any modern programming language

λ-Calculus: History (cont.)

- Is the untyped λ-Calculus consistent? Does it have a semantics? What is its semantic domain D?
  - want "data objects" same as "function objects: since

    $$\lambda x.Mx =_{\eta} M \qquad \Rightarrow D = D^D$$

  - Trouble: impossible unless $|D| = 1$ because $|D| < |D^D| = |D|^{|D|}$
  - Troublesome self-application paradoxes: if we define

    $$\tau = \lambda y.\text{ if } y(y) = a \text{ then } b \text{ else } a$$

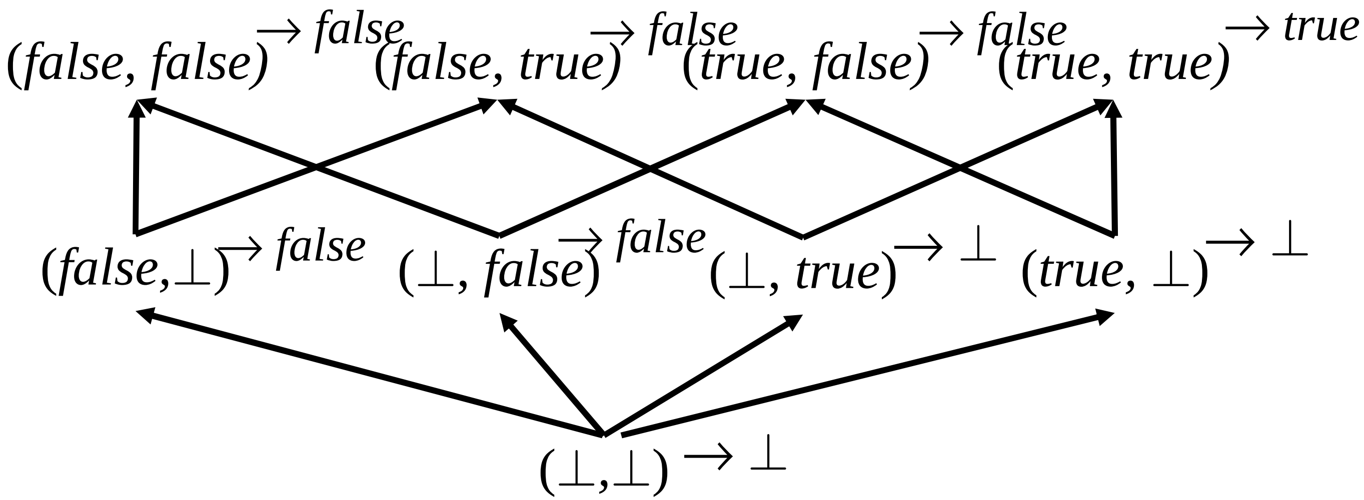
    then $$\tau(\tau) = \text{ if } \tau(\tau) = a \text{ then } b \text{ else } a$$

  - Dana Scott 1970: will work if we have $D = [D \to D]$ the monotone (and continuous) functions of D^D
  - Above example: τ is not monotone

$$\tau(\lambda x.\bot) = \text{ if } \bot = a \text{ then } b \text{ else } a = a$$

$$\tau(\lambda x.a) = \text{ if } a = a \text{ then } b \text{ else } a = b$$

$$\lambda x.\bot \sqsubseteq \lambda x.a \quad \text{but} \quad a \not\sqsubseteq b$$



(false, false) → false
(false, true) → false
(true, false) → false
(true, true) → true
(false,⊥) → false
(⊥, false) → false
(⊥, true) → ⊥
(true, ⊥) → ⊥
(⊥,⊥) → ⊥