Models of Computation

1 Operational Semantics of Expressions

1.1 Big-step Semantics

Big-step, or *natural*, operational semantics ignores the intermediate steps and gives the result immediately.

Deteminacy An expression only evaluates to one value. $\forall E, n_1, n_2.[E \Downarrow n_1 \wedge E \Downarrow n_2 \rightarrow n_1 = n_2]$

Totallity Every expression evaluates to some value. $\forall E.\exists n.[E \Downarrow n]$

1.1.1 Example: definition of B-ADD

$$\text{(B-ADD)}\ \frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = n_1 + n_2$$

where B$-$ADD is a label for human reference.
It means that if E_1 evaluates to n_1, and E_2 evaluates to n_2, then $E_1 + E_2$ evaluates to n_3. So we have $n_3 = n_1 + n_2$.

1.1.2 Example: derivation tree

$$\text{(B-ADD)}\ \frac{\text{(B-ADD)}\ \frac{\text{(B-NUM)}\ \frac{}{4\Downarrow 4} \quad \text{(B-NUM)}\ \frac{}{1\Downarrow 1}}{(4+1)\Downarrow 5} \qquad \text{(B-ADD)}\ \frac{\text{(B-NUM)}\ \frac{}{2\Downarrow 2} \quad \text{(B-NUM)}\ \frac{}{2\Downarrow 2}}{(2+2)\Downarrow 4}}{((4+1)+(2+2)) \Downarrow 9}$$

1.2 Small-step Semantics

Small-step, or *structural*, operational semantics gives a method for evaluating an expression step-by-step.

Deteminacy $\forall E, E_1, E_2\ .[E \rightarrow E_1 \wedge E \rightarrow E_2 \rightarrow E_1 = E_2]$.

Confluence For all E, E_1, E_2, if $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$, then there exists E' such that $E_1 \rightarrow^* E'$ and $E_2 \rightarrow^* E'$.

Weak Normalisation For any expression E_1, there exists a *finite* sequence of expressions $E_2, \ldots, E_k$ such that, E_k is in normal form, and for all $i \in [1..k).E_i \rightarrow E_{i+1}$. (Every expression finally executes to an expression in normal form)

Strong Normalisation There are no infinite sequences of expressions $E_1, E_2, E_3, \ldots$ such that, for all i, $E_1 \rightarrow E_{i+1}$. (Every execution chain terminates)

Unique Normal Form For all E, n_1, n_2, if $E \rightarrow^* n_1$ and $E \rightarrow^* n_2$ then $n_1 = n_2$.

Strong Normalisation implies weak normalisation.

Theorem For all E and n, $E \Downarrow n$ iff $E \rightarrow^* n$.

1.2.1 Examples

$$\text{(S-LEFT)}\ \frac{E_1 \rightarrow E_1'}{E_1 + E_2 \rightarrow E_1' + E_2}$$

$$\text{(S-RIGHT)}\ \frac{E_1 \rightarrow E'}{n + E \rightarrow n + E'}$$

$$\text{(S-ADD)}\ \frac{}{n_1 + n_2 \rightarrow n_3}\ n_3 = n_1 + n_2$$

S−LEFT defines a way to 'simplify' the left operand of the ADD expression.
S−RIGHT does that for the operand on the right-hand-side.
S−ADD defines how to finally evaluate the simplified ADD expression, as all expressions evaluates to a number in this example language.

1.3 Many Steps of Evaluation

$E \rightarrow^* E'$ holds iff either $E = E'$ or there is a finite sequence

$$E \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_k \rightarrow E'$$

The relation $\rightarrow^*$ is called the **reflexive transitive closure** of $\rightarrow$.
Number n is the final answer of E if $E \rightarrow^* n$.

1.4 Normal Form

Normal Form An expression E is in **normal form** (and said to be **irreducible**) if there is no E' such that $E \rightarrow E'$.

Theorem The normal forms of the expressions are the numbers.

2 Operational Semantics of Commands

2.1 States

state is a partial function from variables to numbers such that $s(x)$ is defined for finitely many x.

$s[x \mapsto 7]$ means a new state overriding value of x to 7, and inheriting other values from s.
Example:

$$s = (x \mapsto 4, y \mapsto 5, z \mapsto 6)$$

describes a partial function where variable x has value 4, y has value 5, z has value 6.

$$s[x \mapsto 7](x) = 7$$
$$s[y \mapsto 10](y) = 10$$
$$s[v \mapsto 17](v) = 17$$
$$s[v \mapsto 17](y) = 5$$

2.2 Semantics of The Example Language

2.2.1 Expressions

$$\text{(W-EXP.LEFT)}\ \frac{\langle E_1, s\rangle \rightarrow_e \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \rightarrow_e \langle E_1' + E_2', s'\rangle}$$
$$\text{(W-EXP.RIGHT)}\ \frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle n + E, s\rangle \rightarrow_e \langle n + E', s'\rangle}$$
$$\text{(W-EXP.VAR)}\ \frac{}{\langle x, s\rangle \rightarrow_e \langle n, s\rangle}\ s(x) = n$$
$$\text{(W-EXP.ADD)}\ \frac{}{\langle n_1 + n_2, s\rangle \rightarrow_e \langle n_3, s\rangle}\ n_3 = n_1 + n_2$$

2.2.2 Assignments

$$\text{(W-ASS.EXP)}\ \frac{\langle E, s\rangle \rightarrow_c \langle E', s'\rangle}{\langle x \coloneqq E, s\rangle \rightarrow_c \langle x \coloneqq E', s'\rangle}$$
$$\text{(W-ASS.NUM)}\ \frac{}{\langle x \coloneqq n, s\rangle \rightarrow_c \langle skip, s[x \mapsto n]\rangle}$$

2.2.3 Sequential Composition

$$\text{(W-SEQ.LEFT)}\ \frac{\langle C_1, s\rangle \rightarrow_c \langle C_1^{'}, s^{'}\rangle}{\langle C_1; C_2, s\rangle \rightarrow_c \langle C_1^{'}; C_2, s^{'}\rangle}$$

$$\text{(W-SEQ.SKIP)}\ \frac{}{\langle skip; C_2, s\rangle \rightarrow_c \langle C_2, s\rangle}$$

2.2.4 Conditional

$$\text{(W-COND.TRUE)}\ \frac{}{\langle \text{if true then } C_1 \text{ else } C_2, s\rangle \rightarrow_c \langle C_1, s\rangle}$$

$$\text{(W-COND.FALSE)}\ \frac{}{\langle \text{if false then } C_1 \text{ else } C_2, s\rangle \rightarrow_c \langle C_2, s\rangle}$$

$$\text{(W-COND.BEXP)}\ \frac{\langle B, s\rangle \rightarrow_c \langle B', s'\rangle}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \rightarrow_c \langle \text{if } B' \text{ then } C_1 \text{ else } C_2, s'\rangle}$$

2.2.5 While

$$\text{(W-WHILE)}\ \frac{}{\langle \text{while } B \text{ do } C, s\rangle \rightarrow_c \langle \text{if } B \text{ then } (C; \text{while } B \text{ do } C) \text{ else } skip, s\rangle}$$

2.3 Properties of $\rightarrow_c$

$\rightarrow_c$ is deterministic and confluent.

deterministic : If $\langle C, S\rangle \rightarrow_c \langle C_1', S_1'\rangle$ and $\langle C, S\rangle \rightarrow_c \langle C_2', S_2'\rangle$ then $\langle C_1', S_1'\rangle = \langle C_2', S_2'\rangle$

confluent : If $\langle C, S\rangle \rightarrow_c \langle C_1', S_1'\rangle$ and $\langle C, S\rangle \rightarrow_c \langle C_2', S_2'\rangle$ then $\langle C_1', S_1'\rangle \rightarrow_c \langle C_3', S_3'\rangle$ and $\langle C_2', S_2'\rangle \rightarrow_c \langle C_3', S_3'\rangle$

2.4 Configurations

Configuration A configuration $\langle \text{skip}, s\rangle$ is an **answer configuration**.

Normal Forms A configuration is in normal form if there is no rule for executing its expression. So an *answer configuration* is in normal form.

2.4.1 Stuck Configurations

Stuck Configuration A configuration $\langle y, (x \mapsto 3)\rangle$ is a **stuck configuration** because y cannot be evaluated in the context $(x \mapsto 3)$.

A stuck configuration is also in *normal form* since there is no rule for executing the expression.

$\langle (x \coloneqq y+1), (x \mapsto 3) \rangle$ is stuck because y is not defined in the context.

$\langle (x < y), (x \mapsto 3) \rangle$ is *not yet* stuck, but it reduces to a stuck configuration.

3 Induction over derivations

3.1 Example: Proving by induction

Base case 1

To show

$\forall b \in B.\ [\text{true} \Downarrow b \Rightarrow \text{true} {\rightarrow^*} b]$

Proof

Take $b \in B$ arb.

$$\begin{aligned} \text{true} \Downarrow b & \qquad \textit{ass} \\ b = \text{true} & \qquad \textit{by 1, and} \\ \text{true} {\rightarrow^*} \text{true} & \qquad \textit{->* is reflective} \end{aligned}$$

Base case 2

To show

$\forall b \in B.\ [\text{false} \Downarrow b \Rightarrow \text{false} {\rightarrow^*} b]$

Inductive case 1

Take $B_1, B_2 \in \text{Bool}$,

Inductive Hypothesis

- $\forall b \in B.\ [B_1 \Downarrow b_1 \Rightarrow B_1 {\rightarrow^*} b_1]$
- $\forall b \in B.\ [B_2 \Downarrow b_2 \Rightarrow B_2 {\rightarrow^*} b_2]$

To show

$$\forall b \in B.\ [B_1 \& B_2 \Downarrow b \Rightarrow B_1 \& B_2 {\rightarrow^*} b]$$

...omitted. See Coursework 1 for detailed example.

4 Register Machines

Register Machine is specified by

- finite many registers R_0, R_1, ..., R_n, each capable of storing a natural number.
- a program consisting of a finite list of instructions of the form label : body where, for $i = 0, 1, 2, \ldots$, the $(i+1)^{th}$ instruction has label L_i.

The instruction **body** takes the form:

- $R^+ \to L'$: add 1 to contents of register R and jump to instruction labelled L'.
- $R^- \to L', L''$: if contents of R is > 0, then subtract 1 and jump to L', else jump to L''.
- $HALT$ stop executing instructions.

4.1 Configurations

A register machine **configuration** has the form:

$$c = (l, r_0 \ \ldots \ r_n)$$

where l is current label and r_i is current content of R_i.
Initial configuration is $c_0 = (0, r_0 \ \ldots \ r_n)$

4.2 Computations

A **computation** of a RM is a finite or infinite sequence of configurations:

$$c_0, c_1, c_2, \ldots$$

where c_0 is an initial configuration, others determine next configurations

4.2.1 Halting Computations

A **halting computation** can either be proper or erroneous.
A **proper halt** is when reaching $HALT$. An **erroneous halt** is when reaching an undefined label.

4.3 Partial Functions

A **partial function** from set X to Y is specified by any subset $f \subseteq X \times y$ such that

$$(x, y) \in f \ \wedge (x, y') \in f \ \Rightarrow y = y'$$

4.3.1 Partial Function Notations

- $f(x) = y$ means $(x, y) \in f$
- $f(x) \downarrow$ means $\exists y \in Y(f(x) = y)$
- $f(x) \uparrow$ means $\neg\exists y \in Y(f(x) = y)$
- $X \rightharpoonup Y$ means set of all partial functions from X to Y
- $X \rightarrow Y$ means set of all total functions from X to Y

4.3.2 Total Functions

A **total function** is a partial function that satisfies $\forall x \in X.\ f(x) \downarrow$

4.4 Computable functions

The partial function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is computable is there is a register machine M with at least $n+1$ registers, such that for all $(x_1 \ldots x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:
the computation of M starts with paring of $R_0 = 0, R_1 = x_1, \ldots$ and all other registers set to 0, halts with $R_0 = y$
iff $f(x_1 \ldots x_n) = y$.

4.5 Numerical Codings

4.5.1 Numerical Codings of Pairs

$$\langle\!\langle x,\ y \rangle\!\rangle = 2^x(2 * y + 1)$$
$$< x, y > = 2^x(2 * y + 1) - 1$$

Binary representations:

$$0\text{b}\langle\!\langle x,\ y \rangle\!\rangle = 0\text{b}y\ 1\ 0 \cdots 0$$
$$0\text{b} < x, y > = 0\text{b}y\ 0\ 1 \cdots 1$$

where there are x number of 0s or 1s.

4.5.2 Numerical Coding of Programs

For program P with $L_0 \dots L_n$, $\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner L_0 \urcorner, \dots \ulcorner L_n \urcorner] \urcorner$.

- $\ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\langle 2i,\ j \rangle\rangle$
- $\ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\langle 2i+1,\ < j, k > \rangle\rangle$
- $\ulcorner HALT \urcorner \triangleq 0$

4.5.3 Definition of Lists

An empty list is $[]$.

List is recursively constructed using cons: $x :: l \in \text{List}\mathbb{N}$ if $x \in \mathbb{N}$ and $l \in \text{List}\mathbb{N}$

$[x_1 \dots x_n] \triangleq x_1 :: (x_2 :: (\dots :: (x_n :: [])))$

4.5.4 Numerical Coding of Lists

1. $\ulcorner [] \urcorner \triangleq 0$
2. $\ulcorner x :: l \urcorner \triangleq \langle\langle x,\ \ulcorner l \urcorner \rangle\rangle = 2^x (2 \cdot \ulcorner l \urcorner + 1)$

Thus, $\ulcorner [x_1 \dots x_n] \urcorner = \langle\langle x_1,\ \langle\langle x_2,\ \cdots \langle\langle x_n,\ 0 \rangle\rangle \cdots \rangle\rangle \rangle\rangle$.

Binary representations:

$0\text{b}\ulcorner [x_1 \dots x_n] \urcorner = (1\ 0 \cdots 0)\ (1\ 0 \cdots 0)\ \cdots\ (1\ 0 \cdots 0)$

where each brackets represents an element, the number of zeros in each element is x. Note the first pair is x_n.

4.5.5 Decoding List to Programs

Any $x \in \mathbb{N}$ can be decoded to a body:

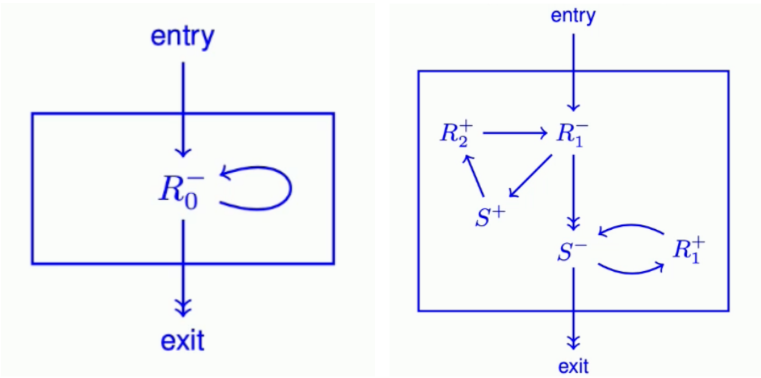
1. If $x = 0$ then body is HALT.
2. Else let $x = \langle\langle y,\ z \rangle\rangle$:
   - If $y = 2i$ is even then body is $R_i^+ \to L_z$
   - If $y = 2i + 1$ is odd, let $z =< j, k >$ then body is $R_i^- \to L_j,\ L_k$

4.6 Gadgets

A **gadget** is a partial register machine graph(subprogram).

It operates on single input, and may output to multiple registers. It can use auxiliary **scratch registers** for its internal use. It assumes scratch registers are initially set to 0 and must restore it to 0 after use.

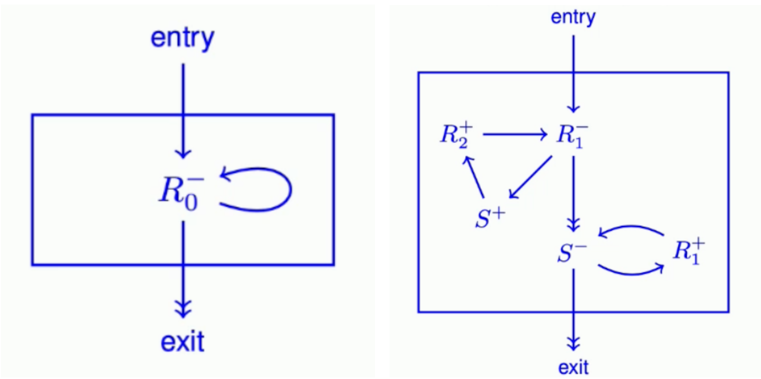
4.6.1 Examples of Gadgets



(a) Zero R_0 **(b)** Add R_1 to R_2

Figure 2: Examples of Gadgets

4.6.2 Usage of Gadgets



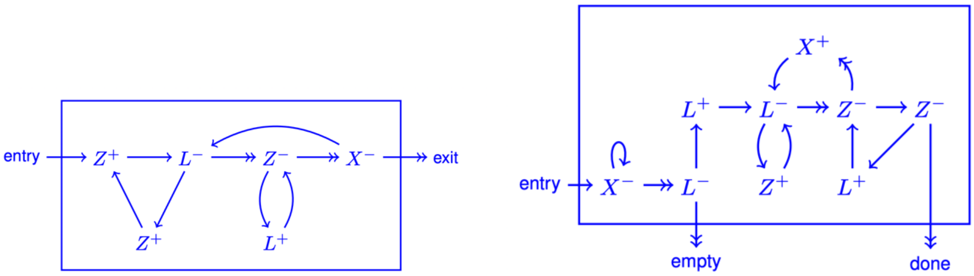
(a) Zero R_0 **(b)** Add R_1 to R_2

Figure 4: Usages of Gadgets

4.7 Lists

Lists can be represented by recursively shifted numbers.

Given input values $X = x$, $L = l$ and $Z = 0$, the push gadget pushes the value x into l. It returns $X = 0$, $L = \langle\langle x,\ l\rangle\rangle = 2^x(2l + 1)$.



(a) Push X to L **(b)** Pop L to X

Figure 6: List Push and Pop

4.8 Universal Register Machine

5 Turing Machines

A **Turing Machine** has a tape and a head which can go left or right on the tape.

It is specified by $M = (Q, \Sigma, s, \delta)$ where

- Q —finite set of machine states.
- Σ —finite set of tape symbols, containing distinguished symbols, *blank*, $\sqcup$
- $s \in Q$ —initial state
- $\delta \in (Q \times \Sigma) \to (Q \times \Sigma \times L, R)$

5.1 Turing Machine Configurations

Turing Machine **configuration** is (q, w, u) consisting of:

- $q \in \mathbb{Q}$ —current state
- $w \in \Sigma^*$ —finite, possibly-empty string of tape symbols to the left of tape head
- $u \in \Sigma^*$ —finite, possibly-empty string of tape symbols to the right of tape head

An **initial configuration** is (s, ϵ, u) for initial state s and string of tape symbols u.

A configuration is in **normal form** if it has no computation step.

5.2 Turing Machine Computation

5.2.1 first and last

Define first and last as follows:

$$\text{first}(w) = \begin{cases} (a, v) & \text{if } w = av \\ (\sqcup, \epsilon) & \text{if } w = \epsilon \end{cases}$$

Note that av means a followed by v instead of mathematical multiplication. The first function retrieves the first symbol of a string, or $\sqcup$ if string is empty.

$$\text{last}(w) = \begin{cases} (a, v) & \text{if } w = va \\ (\sqcup, \epsilon) & \text{if } w = \epsilon \end{cases}$$

last retrives the last symbol of a string, or $\sqcup$ if string is empty.

5.2.2 Turing Machine Computation

If $\delta(q, a) = (q', a', L)$, then the 'head' goes left, $(q, w, u) \to M(q', w', ba'u')$.
If $\delta(q, a) = (q', a', R)$, then the 'head' goes right, $(q, w, u) \to M(q', wa', u)$.

6 Lambda Calculus

$$\begin{aligned} M ::= \ & x && \mathit{Variable} && (1) \\ & \mid \lambda x.M && \mathit{Abstraction} && (2) \\ & \mid M_1 \ M_2 && \mathit{Application} && (3) \end{aligned}$$

Contraction Consecutive abstractions can be contracted: $\lambda x.\lambda y.\lambda z. \equiv \lambda xyz.$

closed term A term is closed if there is no free variable.

6.1 Syntax

6.1.1 Free And Bound Variables

- $FV(x) = x$
- $FV(\lambda x.M) = FV(M)$
  x
- $FV(M \ N) = FV(M) \cup FV(N)$

Examples:

- $\lambda x.x$: x is bound
- $\lambda x.y$: y is free

- $\lambda x.\lambda y.\lambda z.xy$: x and y are bound
- $FV\ ((\lambda x.(\lambda y.x\ y)y)(\lambda z.z\ x)) = x, y$

6.1.2 Association

Application is left-associate. Example:

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

6.1.3 Alpha Equivalence

$$\lambda xy.x\ y{=_\alpha}\lambda ab.a\ b$$

$M{=_\alpha}N$ iff one can be obtained by renaming variables.

6.1.4 Determining Alpha Equivalence

Strategy:

1. Check structure of terms
2. Check free variables match
3. Rename bound variables to check if they match

6.1.5 Substitution

$M[N/x]$ means replace free variable x with N in M.

Substitution cannot either change a free variable into bound nor the reverse.

$$x[N/y] = \begin{Bmatrix} N & x = y \\ x & x \neq y \end{Bmatrix}$$

$$(\lambda x.M)[N/y] = \begin{Bmatrix} \lambda x.M & x = y \\ \lambda z.M[z/x][N/y] & x \neq y \end{Bmatrix}$$

$$(M_1\ M_2)[N/y] = (M_1[N/y])(M_2[N/y])$$

where z is not used in N. i.e. $z \notin (FV(N)\backslash x)$, $z \notin FV(M)$, $z \neq y$.

Examples:

- Replace variable: $x[y/x] \equiv y$

- No matching variable: $z[y/x] \equiv z$
- Replace variable in applications: $(x\ y)(y\ z)[y/x] \equiv (y\ y)(y\ z)$
- Replacing free variable without conflict: $(\lambda z.xz)[y/x] \equiv \lambda z.yz$
- Keep bounded variables bounded: $(\lambda x.xy)[y/x] \equiv \lambda x.xy$
- Keep free variables free: $(\lambda y.xy)[y/x] \equiv \lambda z.yz$
- Rename bounded x: $(\lambda x.xy)[x(\lambda x.xy)/x] \equiv \lambda z.z(x(\lambda x.xy))$

6.2 Semantics

Use β-Reduction to compute λ-Calculus.

Small-step rules:

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

$$\frac{M \to_\beta M'}{M\ N \to_\beta M'\ N}$$

$$\frac{N \to_\beta N'}{M\ N \to_\beta M\ N'}$$

$$\frac{M =_\alpha M' \quad M' \to_\beta N' \quad N' \to_\beta N}{M \to_\beta N}$$

Note β-Reduction is not deterministic, for MN you can choose to reduce M first or N first.

The last rule ensures two equivalent expressions behave in the same way.

6.2.1 Multi-step β-reduction

$$\text{Reflexivity, } \alpha\text{-conversion } \frac{M =_\alpha M'}{M \to_\beta^* M'}$$

$$\text{Transitivity } \frac{M \to_\beta M'' \quad M'' \to_\beta^* M'}{M \to_\beta^* M'}$$

6.2.2 Confluence

Theorem Church-Rosser $\forall M, M_1, M_2. M \to_\beta^* M_1 \land M \to_\beta^* M_2 \Rightarrow \exists M'. M_1 \to_\beta^* M' \land M_2 \to_\beta^* M'$

Theorem Church-Rosser states if M reduces to M_1 or M_2, then M_1 and M_2 both reduces to a M'.

6.2.3 β Normal Forms

λ-terms are in β-normal form if they contain no redexes(i.e. they cannot be reduced further).

Example: $(\lambda x.xx)$ is in normal form because it cannot be reduced.

Uniqueness of β-Normal Forms $\forall M, N_1, N_2 . M \rightarrow^*_\beta N_1 \wedge M \rightarrow^*_\beta N_2 \wedge$

Uniqueness of β-Normal Forms states if M reduces to N_1 or N_2 which are in normal form, then $N_1 =_\alpha N_2$.

Not all λ-terms have a normal form. Like $(\lambda x.x\ x)(\lambda x.x\ x)$ does not have one.

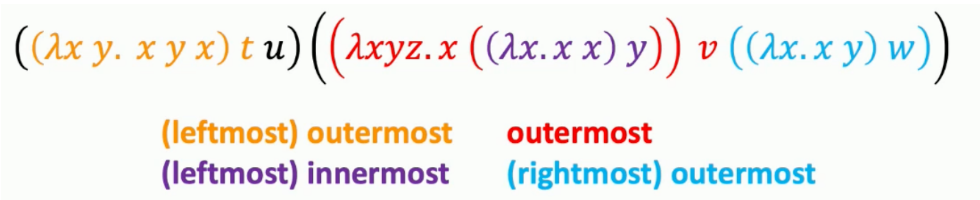
6.2.4 β Equivalent

Two α-terms are β-equivalent($=_\beta$) if they are $=_\alpha$ if applied some β-reductions.

6.2.5 Innermost and Outermost Redexes

For $E = (\lambda x.M)N$:

- Any redex that is in M or in N is inside the redex E.
- Any redex that is in M or in N is inside the redex E.



6.3 Reduction Strategies

Normal Order Reduces the leftmost outermost redex first. Always reduces a term to its normal form if exists.

Not used by any programming languages.

Call By Name Reduces the leftmost outermost redex first. Does not reduce inside λ-abstractions. Not always reduces a term to its normal form.

Examples: Haskell, R, Latex.

Call By Value Reduces the leftmost innermost redex first. Does not reduce inside λ-abstractions. Not always reduces a term to its normal form.

Examples: C, OCaml

$$\big((\lambda x\, y.\ x\ y\ x)\ t\ u\big)\Big(\big(\lambda xyz.x\ ((\lambda x.x\ x)\ y)\big)\ v\ ((\lambda x.x\ y)\ w)\Big)$$

NORMAL ORDER:	first	second	third	fourth (twice)
CALL-BY-NAME:	first	second	never	never
CALL-BY-VALUE:	first	second	third	never

6.4 λ-Definable

A partial function $f : \mathbb{N}^n \to \mathbb{N}$ is **λ-definable** iff there exists a closed λ-term M with:

$$f(x_1, \ldots, x_n) = y \text{ iff } M\ x_1\ \ldots\ x_n =_\beta y$$

and

$$f(x_1, \ldots, x_n)\ \uparrow \text{iff } M\ x_1\ \ldots\ x_n \text{ has no normal form}$$

In short, M reduces to a value iff f computes to the value.

6.5 Church-Turing Thesis

The Church-Turing Thesis f is computable $\leftrightarrow$ f is λ-definable $\leftrightarrow$ f is register-machine-computable $\leftrightarrow$ f is Turing-machine-computable

6.6 Encoding

6.6.1 Encoding Natural Numbers

Church numeral $n \triangleq \lambda f.\lambda x.f(\ldots(f(x))\ldots)$

A Church numeral n means "to do something n times". Example:

- $0 \triangleq \lambda f.\lambda x.x$
- $1 \triangleq \lambda f.\lambda x.f(x)$
- $2 \triangleq \lambda f.\lambda x.f(f(x))$