Models of Computation

WHILE Language. x∈Var ranges over variable identifiers, n∈ℕ ranges over nat

$$B \in \mathit{Bool} ::= \mathtt{true} \mid \mathtt{false} \mid E = E \mid E < E \mid B \& B \mid \neg B \mid \ldots$$
$$E \in \mathit{Exp} ::= x \mid n \mid E + E \mid \ldots$$
$$C \in \mathit{Com} ::= x := E \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \mid C;C \mid \mathtt{skip} \mid \mathtt{while}\ B\ \mathtt{do}\ C$$

1.1) Evaluation Properties

1) Determinacy: We always evaluate to the same result – e.g: if E⇓n_1 and E⇓n_2, n_1=n_2

2) Totality: We always get a result in **normal form**: e.g: for all E, there is n such that E⇓n

3) Reflexive Transitive Closure: →* means 0 or more iterations of →, and n is the final result we get to if E →* n

4) Normal Form: An Expression E is in normal form (and said to be irreducible), if there is no E' such that E -> E'.

5) Confluence: For all E, E_1,E_2, if E→*E_1 and E→*E_2 then there exists E' such that E_1 →*E' and E_2 → *E_1 (AKA there might be several evaluation paths but they all get to the same result).

Determinance → Confluence

6) Strong Normalisation: There are no infinite sequences of expressions E_1 → E_2 → E_3, such that for all i, E_i → E_{i+1}. That is, we always reach a normal form.

1.2) Simple Expressions

E ∈ SimpleExp ::= n | E + E | E x E ...

Normal Form of Expressions: All expressions have a normal form, which are the Natural numbers.

Big Step Semantics: Our rules make us "Leap to the final result". Properties: **Determinacy, Totality.**

Small Step Semantics: "Show the evaluation of each step according to the rules". Properties: (that we must ensure our definitions of small step semantics have): **Determinacy, Confluence, (Strong) Normalisation.**

Link between Big and Small step semantics:

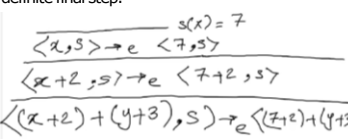
For all E and n, E⇓n if and only if E→*n

1.3) State

We consider state – s = (x → 2, y → 200). It's a variable store. We define our configurations with the form <E, s> <B, s> and <C, s> (a Expr, Boolean or Command and the state).

M1) Derivation Trees

We obtain derivation trees by working from the bottom upwards. We start with the step we want to work out, then find which rule pattern matches it and put it above. We continue until we have a definite final step.



All the other rules are standard, note W-WHILE: Which doesn't actually do any execution, it just unrolls the loop

We have **Determinacy** and **Confluence** for the evaluation of Booleans and Expressions and Commands ($\to_b$, $\to_e$,$\to_c$). Additionally, $\to_b$, $\to_e$ are normalising, but $\to_c$ thanks to the presence of (potentially infinite) while loops.

1.4) The Answer Configuration (ACs)

<n, s> for Expressions. <true, s> or <false, s> for Booleans. <skip, s> for Commands

The format of the final answer of some syntax.

The Link Between NFs, ACs, Stuck Configs:

There are no further rules for evaluating ACs like NFs – **answer configurations are normal forms.** *Normal Forms are not necessarily answer configurations*, we could have <x, s[y→100]>, and since we don't have a value for x, we cannot evaluate further. This example is actually a **Stuck Configuration**. So we're in **Normal Form**, and yet we aren't in an answer configuration.

Normal Forms = ACs + Stuck Configs.

Another example of a Stuck Config:

<x := y + 7; y = x – 1, S = {x |→ 5}>. We can't evaluate the first statement, as we don't know y yet – even if the second one would have determined y. So we're stuck!

1.5) Short Circuit Semantics for Booleans

$$\frac{B_1 \to B_1'}{B_1 \,\&\, B_2 \to B_1' \,\&\, B_2}$$

$$\frac{}{\mathtt{false} \,\&\, B_2 \to \mathtt{false}}$$

$$\frac{}{\mathtt{true} \,\&\, B_2 \to B_2}$$

Strict – an operation is called strict in one of its arguments if it always has to evaluate it. Addition is strict in **both arguments**, for example, and the given short circuit semantics are **left-strict.**

1.6)Procedure/Method Calls Eval Semant.

1) Call By Value (eval all args left to right)
2) Call By Name (evaluate each argument again whenever used)
3) Call By Need (evaluate each argument the first time they're used, and store result for later uses).

2) Structural Induction

Tips:

Consider the possible paths that could have produced this expression (Use **Inversion?**)

Split the proof into cases.

2.1) Inductive Proof Outlines

Base Case Prove that $P(n)$ holds for every number n.

Inductive Case 1 Prove that, for all $E_1$and E_2, $P(E_1 + E_2)$ h… suming the inductive hypotheses that $P(E_1)$ and $P(E_2)$ hold.

Inductive Case 2 Prove $P(E_1 \times E_2)$ similarly.

1.3.2 Multi-step Reductions

Simple induction on numbers. If $P(r)$ is that $E \to^r E'$:

Base Case Prove that $P(0)$ holds.

Inductive Case Prove that, for all k, $P(k+1)$ holds, assuming $P(k)$

1.3.3 Commands

Base Case 1 Prove that $P(\mathtt{skip})$ holds.

Base Case 2 Prove that, for all x and E, $P(x := E)$ holds.

Inductive Case 1 Prove that, for all B, C_a, C_b, $P(\mathtt{if}\ B\ \mathtt{then}\ C_a\ \mathtt{else}\ C_b)$ holds, assuming $P(C_a)$ and $P(C_b)$.

Inductive Case 2 Prove that, for all $C_a$and C_b, $P(C_a;C_b)$ holds, assuming, assuming $P(C_a)$ and $P(C_b)$.

Inductive Case 3 Prove that, for all B and C, $P(\mathtt{while}\ B\ \mathtt{do}\ C)$ holds, assuming, assuming $P(C)$.

1. Induction over the structure of terms.
If we are proving something of the form $\forall E \in Exp.Q(E)$ or $\forall C \in Com.Q(C)$ then I generally use induction over the structure of the Expression or Command respectively. For example, if we have something like $Q(E) = \exists n \in \mathbb{N}.([[E]] = n)$. The inductive principle would look like:

$$\begin{aligned} & \forall n \in \mathbb{N}.Q(n) \\ \wedge\ & \forall E_1, E_2 \in Exp.[Q(E_1) \wedge Q(E_2) \Rightarrow Q(E_1 + E_2)] \\ \wedge\ & \ldots \\ \Longrightarrow\ & \forall E \in Exp.[Q(E)] \end{aligned}$$

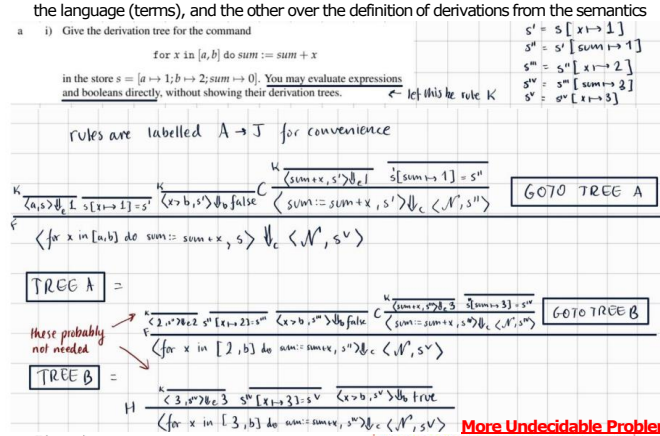
2. Induction over the structure of derivation.
If we are proving something of the form $\forall E, E' \in Exp.[E \Downarrow_e E' \implies Q(E, E')]$, then I would generally use induction over the derivation. The exact derivation type $(\to, \Downarrow, \rightsquigarrow)$ does not matter. The inductive principle (for the big step semantics) looks like:

$$\begin{aligned} & \forall n \in \mathbb{N}.[Q(n,n)] \\ \wedge\ & \forall E_1, E_2 \in Exp. \forall n_1, n_2, n_2 \in \mathbb{N}.[Q(E_1, n_1) \wedge Q(E_2, n_2) \wedge n_3 = n_1 + n_2 \Rightarrow Q(E_1 + E_2, n_3)] \\ \wedge\ & \ldots \\ \Longrightarrow\ & \forall E, E' \in Exp.[E \Downarrow_e E' \implies Q(E, E')] \end{aligned}$$

3. Induction over multiple steps.
Finally, if we are proving something of the form $\forall E, E' \in Exp.[E \to_e^* E' \implies Q(E, E')]$ then I would use induction over multiple steps. This is actually is a specialisation of induction over a transitive closure (see notes). The inductive principle looks like:

$$\begin{aligned} & \forall E \in Exp.[Q(E,E)] \\ \wedge\ & \forall E, E', E'' \in Exp.[E \to E'' \wedge Q(E'', E') \implies Q(E, E')] \\ \Longrightarrow\ & \forall E, E' \in Exp.[E \to * E' \implies Q(E, E')] \end{aligned}$$

Inducting backwards is sometimes necessary! Big step semantics instantly leap to the answ… so we may need some sort of auxiliary precondition to let us leap there. We can have Structural Induction over Terms of something, and we could have induction over the Struct… of Derivations. There is a clear difference - for one of them we induct over the definitions o… the language (terms), and the other over the definition of derivations from the semantics



$P(\mathtt{true})$ ∧ $P(\mathtt{false})$ ∧ $\forall B_1, B_2 \in \mathit{Bool}.[P(B_1) \wedge P(B_2) \to P(B_1 \& B_2$ ∧ $\forall B \in \mathit{Bool}.[P(B) \Downarrow P(\neg B)]$ $\Longrightarrow$ $\forall B \in \mathit{Bool}.[P(B)]$

For each $e \in \mathbb{N}$, let $\varphi_e \in \mathbb{N} \to \mathbb{N}$ be the unary partial function computed by the RM with program $prog(e)$. So for all $x, y \in \mathbb{N}$:

$\varphi_e(x) = y$ holds iff the computation of $prog(e)$ started with $R_0 = 0, R_1 = x$ and all other registers zeroed eventually halts with $R_0 = y$.

Thus

$$e \mapsto \varphi_e$$

defines an **onto** function from $\mathbb{N}$ to the collection of all computable partial functions from $\mathbb{N}$ to $\mathbb{N}$.

An uncomputable function

Let $f \in \mathbb{N} \to \mathbb{N}$ be the partial function $\{(x, 0) \mid \varphi_x(x)\uparrow\}$.

Thus $f(x) = \begin{cases} 0 & \text{if } \varphi_x(x)\uparrow \\ \textit{undefined} & \text{if } \varphi_x(x)\downarrow \end{cases}$

f is not computable, because if it were, then $f = \varphi_e$ for some $e \in \mathbb{N}$ and hence

- if $\varphi_e(e)\uparrow$, then $f(e) = 0$ (by def. of f); so $\varphi_e(e) = 0$ (by def. of e), i.e. $\varphi_e(e)\downarrow$
- if $\varphi_e(e)\downarrow$, then $f(e)\uparrow$ (by def. of e); so $\varphi_e(e)\uparrow$ (by def. of f)

Contradiction! So f cannot be computable.

Models of Comp II

3) Register Machines

Operate with Natural Nums on finitely many registers. Turing Complete. Instructions:

1) R^+ → L_i – Add 1 to R and jump to L.
2) R^- → L_i, L_j. If R > 0, subtract 1 and jump to L_i, else jump to L_j
3) HALT – stop executing instructions

Code Representation: Each Instruction consists if a label (line number) and a body (one of 3 instr above).

A register machine configuration has the form: C = (l, r_0, ... r_n) – standing for label and then all the register contents. R_i = x in the config c mean the ith register of C has value x.

Graph Representation: **Don't Forget START →**

3.1) True and Erroneous Halts

For a finite computation (C_1,C_2,... C_m), the halting configuration is denoted as C_m = (l, r_1 ... r_n). If l is HALT, then we have a **Proper Halt** otherwise if we jump to some instruction which doesn't exist then we have an **Erroneous Halt.**

Computation is **deterministic**: relation between initial and final register contents is a partial function (loops forever if undefined for a given input).

3.2) Computable Functions

Definition. The partial function $f \in \mathbb{N}^n \to \mathbb{N}$ is **(register machine) computable** if there is a register machine M with at least $n+1$ registers $R_0, R_1, \ldots, R_n$ (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$,

the computation of M starting with $R_0 = 0, R_1 = x_1, \ldots, R_n = x_n$ and all other registers set to 0, halts with $R_0 = y$

if and only if $f(x_1, \ldots, x_n) = y$.

3.3) The Halting Problem

The Halting Problem is the decision problem with

- the set S of all pairs (A, D), where A is an algorithm and D is some input datum on which the algorithm is designed to operate;
- the property $A(D)\downarrow$ holds for $(A, D) \in S$ if algorithm A when applied to D eventually produces a result: that is, eventually halts

Turing and Church's work shows that the Halting Problem is unsolvable (undecidable): that is, there is no algorithm H such that, for all $(A, D) \in S$,

$$H(A, D) = \begin{cases} 1 & A(D)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

The Halting problem is Undecidable (Proof):

Assume that we have H such that H(A, D) = 1, if A(D) – that is the algorithm A on input D halts (↓) return 1, or 0 otherwise (↑)

First Construct C such that "given input A, compute H(A, A); if H(A, A) = 0 (halts) then return 1 else loop forever".

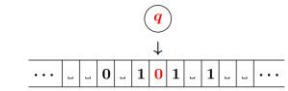
1) Then ∀A: (C(A)↓ ↔ H(A, A) = 0)
2) ∀A: (H(A, A) = 0 ↔ ¬A(A)↓)
3) ∀A: (C(A) = 0 ↔ ¬A(A)↓)
4) Take A to be C: C(C)↓ ↔ ¬C(C)↓, contradiction, we don't have a machine which decides the halting problem – it's undecidable.

Explanation:

C is H with a modified output - if H decided our program halts then return 1 or loop forever.

1) Feed arb program A into C. If C(A) halts, then we know H(A, A) doesn't halt, and returns 0.
2) Because H(A, A) = 0, we know ¬A(A) by definition of H(A, A).
3) So, C(A) implies A doesn't halt.
4) Then just take C as A and we have contradiction!

4) Turing Machines



- The machine starts in state s with the tape head pointing to the first symbol of the finite input string. (Everything to the left and right of the input string is initially blank.)
- The machine computes in steps, each depending on the current state (q) and symbol being scanned by tape head (0)
- An action at each step is to: overwrite the current tape cell with a symbol; move left or right one cell; and change state.

A **Turing machine** is specified by a quadruple $M = (Q, \Sigma, s, \delta)$ where

- Q is a finite set of machine states;
- Σ is a finite set of tape symbols, containing distinguished symbol ␣, called blank;
- an initial state $s \in Q$;
- a partial transition function

$$\delta \in (Q \times \Sigma) \to (Q \times \Sigma \times \{L, R\})$$

A Turing Machine configuration (q, w, u) consists of

- the current state $q \in Q$;
- a finite, possibly-empty string $w \in \Sigma^*$ of tape symbols to the left of tape head;
- a finite, possibly empty string $u \in \Sigma^*$ of tape symbols under and to the right of tape head. ϵ denotes the empty string.

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

The configuration only describes the contents of tape cells that are part of the input or have been visited by the Turing machine. Everything else is blank.

first and last

Define the functions **first** : $\Sigma^* \to \Sigma \times \Sigma^*$ and **last** : $\Sigma^* \to \Sigma \times \Sigma^*$ as follows

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

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

Given $M = (Q, \Sigma, s, \delta)$, define $(q, w, u) \to_M (q', w', u')$ by

$$\frac{\textbf{first}(u) = (a, u') \quad \delta(q, a) = (q', a', L) \quad \textbf{last}(w) = (b, w')}{(q, w, u) \to_M (q', w', ba'u')}$$

$$\frac{\textbf{first}(u) = (a, u') \quad \delta(q, a) = (q', a', R)}{(q, w, u) \to_M (q', wa', u')}$$

We say that a configuration (q, w, u) is a normal form if it has no computation step. This is the case exactly when $\delta(q, a)$ is undefined for $\textbf{first}(u) = (a, u')$.

Consider the TM $M = (Q, \Sigma, s, \delta)$ where $Q = \{s, q, q'\}$ $\Sigma = \{\text{␣}, 0, 1\}$ and the transition function $\delta \in (Q \times \Sigma) \to (Q \times \Sigma \times \{L, R\})$ is given by:

δ	␣	0	1
s	$(q, \text{␣}, R)$		
q	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'			$(q', 1, L)$

Theorem: The computation of a Turing machine M can be implemented by a register machine.

Lists as Tape:

Definition. A tape over $\Sigma = \{\text{␣}, 0, 1\}$ **codes a list of numbers** i… precisely two cells contain **0** and the only cells containing **1** occur between these.

Such tapes look like:

$$\underbrace{\cdots \text{␣}}_{\text{all ␣'s}} 0 \underbrace{1 \cdots 1}_{n_1} 0 \underbrace{1 \cdots 1}_{n_2} 0 \cdots 0 \underbrace{1 \cdots 1}_{n_k} 0 \underbrace{\text{␣} \cdots}_{\text{all ␣'s}}$$

which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

Note the blank spaces: ␣!

4) Numerical Enc. of Pairs, Lists, Programs

4.1) Pairs

Definition

For $x, y \in \mathbb{N}$, define $\begin{cases} \langle\!\langle x, y \rangle\!\rangle \triangleq 2^x(2y+1) \\ \langle x, y \rangle \triangleq 2^x(2y+1) - 1 \end{cases}$

Example $27 = 0\text{b}11011 = \langle\!\langle 0, 13 \rangle\!\rangle = \langle 2, 3 \rangle$

Result

$\langle\!\langle -, - \rangle\!\rangle$ gives a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}^+ = \{n \in \mathbb{N} \mid n \neq 0\}$.

$\langle -, - \rangle$ gives a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

Recall the definition of bijection from discrete maths.

Proof of bijection:

It is enough to observe that

$$\boxed{0\text{b}\langle\!\langle x, y \rangle\!\rangle} = \boxed{0\text{b}y}\boxed{1}\boxed{0\cdots0} \quad x \text{ number of 0s}$$

$$\boxed{0\text{b}\langle x, y \rangle} = \boxed{0\text{b}y}\boxed{0}\boxed{1\cdots1} \quad x \text{ number of 1s}$$

where $0\text{b}x \triangleq x$ in binary. $\triangleq$ means 'is defined to be'.

4.2) Lists

Let $List\,\mathbb{N}$ be the set of all finite lists of natural numbers, defined

- **empty list**: $[\,]$
- **list cons**: $x :: \ell \in List\,\mathbb{N}$ if $x \in \mathbb{N}$ and $\ell \in List\,\mathbb{N}$

Notation: $[x_1, x_2, \ldots, x_n] \triangleq x_1 :: (x_2 :: (\cdots x_n :: [\,] \cdots))$

For $\ell \in List\,\mathbb{N}$, define $\ulcorner \ell \urcorner \in \mathbb{N}$ by induction on the length of the list ℓ:

$$\begin{cases} \ulcorner [\,] \urcorner \triangleq 0 \\ \ulcorner x :: \ell' \urcorner \triangleq \langle\!\langle x, \ulcorner \ell' \urcorner \rangle\!\rangle = 2^x(2 \cdot \ulcorner \ell' \urcorner + 1) \end{cases}$$

Examples

$\ulcorner [3] \urcorner = \ulcorner 3 :: [\,] \urcorner = \langle\!\langle 3, 0 \rangle\!\rangle = 2^3(2 \cdot 0 + 1) = 8$

$\ulcorner [1, 3] \urcorner = \langle\!\langle 1, \ulcorner [3] \urcorner \rangle\!\rangle = \langle\!\langle 1, 8 \rangle\!\rangle = 34$

$\ulcorner [2, 1, 3] \urcorner = \langle\!\langle 2, \ulcorner [1, 3] \urcorner \rangle\!\rangle = \langle\!\langle 2, 34 \rangle\!\rangle = 276$

4.3) Programs

If P is the RM program $\begin{vmatrix} L_0 : body_0 \\ L_1 : body_1 \\ \vdots \\ L_n : body_n \end{vmatrix}$ then its numerical code is

$$\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner body_0 \urcorner, \ldots, \ulcorner body_n \urcorner] \urcorner$$

where the numerical code $\ulcorner body \urcorner$ of an instruction body is defin

by: $\begin{cases} \ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\!\langle 2i, j \rangle\!\rangle \\ \ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\!\langle 2i+1, \langle j, k \rangle \rangle\!\rangle \\ \ulcorner HALT \urcorner \triangleq 0 \end{cases}$

Any $x \in \mathbb{N}$ decodes to a unique instruction $body(x)$:

if $x = 0$ then $body(x)$ is $HALT$,

else ($x > 0$ and) let $x = \langle\!\langle y, z \rangle\!\rangle$ in

if $y = 2i$ is even, then $body(x)$ is $R_i^+ \to L_z$,

else $y = 2i + 1$ is odd, let $z = \langle j, k \rangle$ in

$body(x)$ is $R_i^- \to L_j, L_k$

So any $e \in \mathbb{N}$ decodes to a unique program $prog(e)$, called the register machine **program with index** e:

$$prog(e) \triangleq \begin{vmatrix} L_0 : body(x_0) \\ \vdots \\ L_n : body(x_n) \end{vmatrix} \text{ where } e = \ulcorner [x_0, \ldots, x_n] \urcorner$$

Example of $prog(e)$

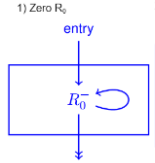
- $786432 = 2^{19} + 2^{18} = 0\text{b}11\underbrace{0\ldots0}_{18\text{ "0"s}} = \ulcorner [18, 0] \urcorner$
- $18 = 0\text{b}10010 = \langle 1, 4 \rangle = \langle 1, \langle 0, 2 \rangle \rangle = \ulcorner R_0^- \to L_0, L_2 \urcorner$
- $0 = \ulcorner HALT \urcorner$

So $prog(786432) = \begin{vmatrix} L_0 : R_0^- \to L_0, L_2 \\ L_1 : HALT \end{vmatrix}$

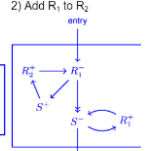
5) Gadgets

Partial Reg Machine graph, with one entry wire and one or more exit. Operates on input and output regs specified in gadget name, but may also use other scratch registers **(YOU MUST ZERO THEM)**. Scratch regs assumed to be set to 0 initially.

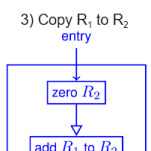
1) Zero R_0



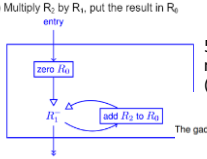
2) Add R_1 to R_2



3) Copy R_1 to R_2



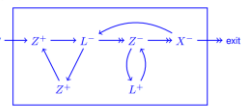
5) Multiply R_0 by R_1, put the result in R_6



5) Push, Push and Pop work by removing or adding an item to a list (numerical encoding of lists).

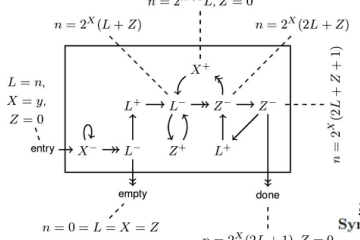
Gadget: "push X to L"

The gadget "push X to L":



6) Pop

Given input values $X = x$, $L = \ell$ and $Z = 0$, it returns the output



Note that the first bit, until L+, is basically just doing some checks, and establishing some Preconditions. After that, we basically get to the inverse of the push machine. (Ie, the triangle subtracts 1 from L and then halves). We increase X each time we do this.

6) The Universal Register Machine:

The Universal Register Machine

The *universal register machine* carries out the following computation, starting with $R_0 = 0$, $R_1 = e$ (code of a program), $R_2 = a$ (code of a list of arguments) and all other registers zeroed:

- decode e as a RM program P
- decode a as a list of register values $a_1, \ldots, a_n$
- carry out the computation of the RM program P starting with $R_0 = 0, R_1 = a_1, \ldots, R_n = a_n$ (and any other registers occurring in P set to 0).

Mnemonics for the registers of U and the role they play:

R_0 = result of simulated RM computation (if any)
R_1 = Program code of RM to be simulated,
R_2 = list of RM arguments R_3 = PC
R_5 = label number of next instr – also holds c ode of curr instr.
R_5 = code of curr instruction body.
R_6 = value of register to be used by current instruction. R_7 and R_8 are auxiliary registers, R_9 onwards are scratch registers.

Overall structure of the URM

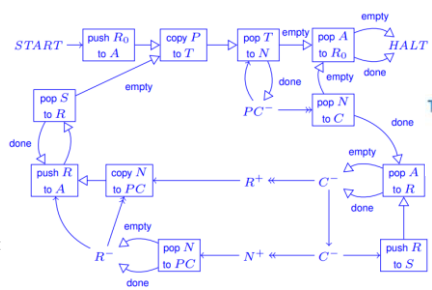
1 copy PCth item of list in P to N (halting if $PC >$ length of list); goto 2

2 if $N = 0$ then halt, else decode N as $\langle y, z \rangle$; $C ::= y$; $N ::= z$; goto 3

{at this point either $C = 2i$ is even and current instruction is $R_i^+ \to L_z$, or $C = 2i + 1$ is odd and current instruction is $R_i^- \to L_j, L_k$ where $z = \langle j, k \rangle$}

3 copy ith item of list in A to R; goto 4

4 execute current instruction on R; update PC to next label; restore register values to A; goto 1



7) Turing Computability:

Definition. $f \in \mathbb{N}^n \to \mathbb{N}$ is **Turing computable** if and only if there is a Turing machine M with the following property:

Starting M from its initial state with tape head on the leftmost 0 of a tape coding $[x_1, \ldots, x_n]$, M halts if and only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

Theorem. A partial function is Turing computable if and only if it is register machine computable.

Proof (sketch). We've seen how to implement any TM by a RM. Hence f TM computable implies f RM computable.

For the converse, one has to implement the computation of a RM in terms of a TM operating on a tape coding RM configurations. To do this, one has to show how to carry out the action of each type of RM instruction on the tape. It should be reasonably clear that this is possible in principle, even if the details are omitted (because they are tedious).

8) Lambda Calculus:

Syntax

- **Bound Variables** x is bound inside $\lambda x\,.\,M$ (it is bound within the scope of M)
- **Free Variables** y is free inside $\lambda x\,.\,M$ (it is not bound)
- **Closed Term** A λ-term with no free variables, e.g $\lambda x\,y\,z\,.\,x\,y$
- **Binding Occurences** The λ-term's parameters $\lambda x\,y\,z\,.\,(\ldots)$, here the x, y and z before the dot.
- **Left Associativity** Lambda Terms are left associative, hence $A\,B\,C\,D \equiv ((((A)\,(B))\,(C))\,(D)$

Bound and Free Formally

$$\begin{aligned} FreeVariables\ (x) &= \{x\} \\ FreeVariables\ (\lambda x\,.\,M) &= FreeVariables(M) \setminus \{x\} \\ FreeVariables\ (M\,N) &= FreeVariables(M) \cup FreeVariables(N) \end{aligned}$$

Renaming: When we have a lambda calculus term, we can rename a variable with a different name in all its occurrences, and ensure that it is the same. We might want to do this to prevent name clashes between operations.

α-equivalence

M =α N iff N can be obtained from M by renaming bound variables, or vice versa. The free variable set MUST NOT BE CHANGED (not even renamed).

STRATEGY: Are the terms of the same structure?
Do all of the free variables match?
Can you rename the bound variables so that they match?

Substitution

$$x[M/y] = \begin{cases} M & x = y \\ x & x \neq y \end{cases}$$

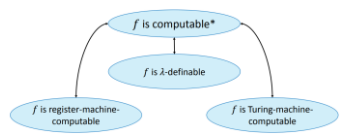
$$(\lambda x\,.\,N)[M/y] = \begin{cases} \lambda x\,.\,N & x = y \text{ (x will be bound inside, so cannot go further)} \\ \lambda z\,.\,N[z/x][M/y] & x \neq y \text{ (To avoid name conflicts with } M, z \notin ((FV(N) \setminus \{x\}) \cup FV(M)) \cup \{y\}) \end{cases}$$

$$(A\,B)[M/y] = (A[M/y])\,(B[M/y])$$

You cannot swap a bound variable out, and you can't swap in a variable that was bound.

(a)	$(xy)[z/x] = zy$	Simple substitution.
(b)	$(xy)[\lambda x.\,xx/x] = (\lambda x.\,xx)y$	Simple substitution.
(c)	$(\lambda x.\,xy)[z/y] = \lambda x.\,xz$	Substitution under binder, no capture threat.
(d)	$(\lambda x.\,xy)[z/x] = \lambda x.\,xy$	No free x to be substituted, no effect.
(e)	$(\lambda x.\,xy)[x/y] = \lambda z.\,zx$	Must rename binding x to avoid capture.
(f)	$(\lambda x.\,xx)[\lambda x.\,xx/x] = \lambda x.\,xx$	No free x to be substituted, no effect.
(g)	$(\lambda x.\,xy)[\lambda x.\,xy/y] = \lambda x.\,x(\lambda x.\,xy)$	Substitution under binder, no capture threat.
(h)	$(\lambda x.\,xy)[x(\lambda x.\,xy)/y] = \lambda z.\,z(\lambda x.\,xy))$	Must rename binding x to avoid capture.

Church Turing Thesis



B-Reduction

Lambda calculus is confluent (always arrives at same result), reflexive, transitive.

λ-terms are in normal form if it has no redex. Normal forms are unique:

Theorem (Uniqueness of β-normal forms)

$$\forall M, N_1, N_2.\ M \to_\beta^* N_1 \wedge M \to_\beta^* N_2 \wedge \text{is\_in\_nf}(N_1) \wedge \text{is\_in\_nf}(N_2) \implies N_1 =_\alpha N_2$$

Not all terms have Normal Forms! They don't ever reduce to an irreducible – makes sense as lambda calc can implement turing machines which suffer the halting problem. E.g: a lambda calculus term which reduces to itself (λx.x x) (λx.x x)

Innermost Redex A Redex with no redexes inside of

Outermost Redex A Redex with no redexes outside

Reduction Strategies

1. **Normal order.** Reduce leftmost-outermost redex first.
2. **Call by name.** Reduce leftmost-outermost redex first, but do not reduce inside λ-abstractions. (Evaluates arguments later).
3. **Call by value.** Reduce leftmost-innermost redex first, but do not reduce inside λ-abstractions. (Evaluates arguments first).

λ-Definable Functions

- $f \in \mathbb{N}^n \to \mathbb{N}$ is λ-definable if there is a closed λ-term F that represents it, such that:
  - If $f(x_1, \ldots, x_n) = y$ then $Fx_1 \ldots x_n =_\beta y$.
  - If $f(x_1, \ldots, x_n)\uparrow$ then $Fx_1 \ldots x_n$ has no β-normal form.
- A function is computable iff it is λ-definable.

Computability

Register Machines, Turing Machines: Computability

RM-computability. A partial function $f : \mathbb{N}^n \to \mathbb{N}$ is RM-computable *iff* there exists a register machine M with at least $n+1$ registers, $R_0, \ldots, R_n$, with the following property: starting M from the state in which $R_0 = 0$, $R_i = x_i\ \forall_{i=1}^n$, M halts *iff* $f(x_1, \ldots, x_n)\downarrow$, and in that case $R_0 = y$, where $y = f(x_1, \ldots, x_n)$.

Turing-computability. A partial function $f : \mathbb{N}^n \to \mathbb{N}$ is Turing-computable *iff* there exists a Turing machine M with the following property: starting M from its initial state, with tape head on the leftmost 0 of a tape coding $[x_1, \ldots, x_n]$, M halts *iff* $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list whose first element is y, where $y = f(x_1, \ldots, x_n)$.

The minsky and successor machines can build each other. Explain why this supports the Church-Turing thesis. The Church-Turing thesis is that anything that is algorithmically computable is computable by a Turing machine.

Anything that is computable by a Minsky machine is computable by a Turing machine, and hence anything that is computable by a Successor machine is also computable by a Turing machine. The Successor machine can reasonably be seen as a formalisation of an algorithm, which could be implemented. If it could compute anything that is not Turing-computable, it would falsify the Church-Turing thesis. The fact that it cannot therefore supports the thesis.

Church Numerals

1) We represent natural numbers as Church Numerals – n repeated applications of our function f: $\underline{n} \triangleq \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{n \text{ times}}$

2) Addition: Since n is the application of f n times to x, and m is the application of f m times to x, n + m is obtained by: plus $= \lambda mnfx.m\ f\ (n\ f\ x)$

3) Multiplication:

$$\texttt{mult} \equiv \lambda m.\lambda n.\lambda f.\ m\ (n\ f)$$

$$\underline{m} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{m \text{ times}} \quad \underline{n} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{n \text{ times}}$$

$$\underline{m \times n} \triangleq \underbrace{(\lambda m\,.\,\lambda n\,.\,\lambda f\,.\,m\ (n\ f))}_{\times}\ \underline{m}\ \underline{n}$$

4) Exponential

$$\underline{m} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{m \text{ times}} \quad \underline{n} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{n \text{ times}}$$

$$\underline{m^n} \triangleq \underbrace{(\lambda m\,.\,\lambda n\,.\,n\ m)}_{\text{exponential}}$$

5) Conditional:

$$\underline{m} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{m \text{ times}}$$

$$\text{if } m = 0 \text{ then } x_1 \text{ else } x_2 \triangleq \underbrace{(\lambda m\,.\,\lambda x_1\,.\,\lambda x_2\,.\,m\ (\lambda z\,.\,x_2)\ x_1)}_{\text{if zero}}$$

If $\underline{m} = 0 = \lambda f\,.\,\lambda x\,.\,x$ then x is returned, which will be x1. If not zero, then the f applied returns x2, so any number of applications of f, results in x2.

7.3 Semantics

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

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

- A term of the form $(\lambda x\,.\,N)\ M$ is called a *redex*.
- A λ-term may have several different reductions. These different reductions for a *derivation tree*.

7.3.1 Multi-Step Reductions

Steps can be combined using the transitive closure of $\to_\beta$ under α-conversion.

$$\frac{M =_\alpha M'}{M \to_\beta^* M'} \quad \text{(Reflexivity of α-conversion)}$$

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

η-equivalence — **Definition 7.**

Captures equality better than $=_\beta$.

$$\frac{x \notin FV(M)}{\lambda x\,.\,M\ x =_\eta M} \quad \frac{\forall N.\ M\ N =_{\eta^+} M'\ N}{M =_{\eta^+} M'}$$

Namely if the application of M to another λ-term is equivalent to M' applied to the same λ-terms then M and M' are equivalent.

For example with the basic application of f:

$$\lambda x\,.\,f\ x \neq_\beta f \quad \text{however} \quad (\lambda x\,.\,f\ x)\ M =_\beta f\ M \quad \text{and} \quad \lambda x\,.\,f\ x \neq_\eta f$$

6) Successor: We simply take m and apply f one more time

$$\underline{m} = \lambda f\,.\,\lambda x\,.\,\underbrace{f(\ldots(f\ x)\ldots)}_{m \text{ times}} \quad \underline{m+1} \triangleq \underbrace{(\lambda m\,.\,\lambda f\,.\,\lambda x\,.\,f\ (m\ f\ x))}_{\text{succ}}\ \underline{m}$$

7) Pairs: We can encode pairs as a function, with a selector s function. Hence by supplying first or second as the selector, we can use the pair.

$$newpair(a, b) \triangleq \underbrace{(\lambda a\,.\,\lambda b\,.\,\lambda s\,.\,s\ a\ b)}_{\text{newpair}}\ a\ b \equiv \underbrace{(\lambda a\ b\ s\,.\,s\ a\ b)}_{\text{newpair}}\ a\ b$$

$$first(p) \triangleq p\ \underbrace{(\lambda x\,.\,\lambda y\,.\,x)}_{\text{first}} \equiv p\ \underbrace{(\lambda x\ y\,.\,x)}_{\text{first}}$$

$$second(p) \triangleq p\ \underbrace{(\lambda x\,.\,\lambda y\,.\,y)}_{\text{second}} \equiv p\ \underbrace{(\lambda x\ y\,.\,y)}_{\text{second}}$$

8) $$pred(n) \triangleq \underbrace{(\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,n\ (\lambda g\,.\,\lambda h\,.\,h\ (g\ f))\ (\lambda u\,.\,x)\ (\lambda u\,.\,u))}_{\text{predecessor}}\ \underline{n}$$

9) $$\underline{m - n} \triangleq \underbrace{(\lambda m\,.\,\lambda n\,.\,m\ pred\ n)}_{\text{subtract}}\ \underline{m}\ \underline{n}$$

Combinators

A closed λ-term (no free variables), usually denoted by capital letters

I	$\triangleq \lambda x\,.\,x$	$I(x)$	$\triangleq x$
K	$\triangleq \lambda x\,y\,.\,x$	$K(x,y)$	$\triangleq x$
S	$\triangleq \lambda x\,y\,z\,.\,x\,z\,(y\,z)$	$S(x,y,z)$	$\triangleq x(z)(y(z))$
T	$\triangleq \lambda x\,y\,.\,y\,x$	$T(x,y)$	$\triangleq y(x)$
C	$\triangleq \lambda x\,y\,z\,.\,x\,z\,y$	$C(x,y,z)$	$\triangleq x(z)(y)$
V	$\triangleq \lambda x\,y\,z\,.\,z\,x\,y$	$V(x,y,z)$	$\triangleq z(x)(y)$
B	$\triangleq \lambda x\,y\,z\,.\,x\,(y\,z)$	$B(x,y,z)$	$\triangleq x(y(z))$
B'	$\triangleq \lambda x\,y\,z\,.\,y\,(x\,z)$	$B'(x,y,z)$	$\triangleq y(x(z))$
W	$\triangleq \lambda x\,y\,.\,x\,y\,y$	$W(x,y)$	$\triangleq x(y)(y)$
Y	$\triangleq \lambda g\,.\,(\lambda x\,.\,g\ (x\ x))\ (\lambda x\,.\,g\ (x\ x))$	$Y(f)$	$\triangleq (\lambda x \to f(x(x)))(\lambda x \to f(x(x)))$

Only SKI are required to define any computable function (can remove even λ-abstraction, this is called SKICombinator Calculus). The Y - Combinator is used for recursion. In one step of β-reduction: Y f →β f (Y f) We cannot define λ-terms in terms of themselves, as the λ-term is not yet defined, and infinitely large λ-terms are not allowed.

Fixed-Point Combinator

A higher order function (e.g fix) that returns some function of itself:

$$fix\ f = f(fix\ f)$$

$$fix\ f = f(f(\ldots f(fix\ f)\ldots)) \text{ (after repeated application)}$$

Factorial — **Example**

$$fact(n) = \begin{cases} 1 & n = 0 \\ n \times fact(n-1) & otherwise \end{cases}$$

If recursive definitions for λ-terms were allows, we could express this as:

$$fact \triangleq \lambda n\ .\ \text{if zero } n\ \underline{1}\ (\underline{multiply}\ n\ (fact\ (pred\ n)))$$

$$\triangleq (\lambda f\ .\ \lambda n\ .\ \text{if zero } n\ \underline{1}\ (\underline{multiply}\ n\ (f\ (pred\ n))))\ fact$$

Since we can use the above form (higher order function applied to itself) with the Y combir

$$fact \triangleq Y(\lambda f\ .\ \lambda n\ .\ \text{if zero } n\ \underline{1}\ (\underline{multiply}\ n\ (f\ (pred\ n))))$$