MoC

Finite: Description of the procedure in terms of elementary operations.
Deterministic: If there is a next step, it is uniquely determined - that is on the same data, the same steps will be made.

Denotational Semantics: a program's meaning is described compositionally using **mathematical objects** (called denotations). A denotation of a program phase is built from denotations of its sub-phrases
Operational Semantics: a program's meaning is given in terms of the **steps of computation** made when it runs

The **Halting problem** is a decision problem with:
- The set of all pairs (A, D) such that A is an algorithm and D is some input datum on which the algorithm operates
- The property A(D) ↓ holds for (A, D) ∈ S if an algorithm A when applied to D produces a result (halts)

Turing and Church showed that there is no algorithm such that:

$$\forall (A,D) \in S \begin{bmatrix} H(A,D) & = & 1 & A(D)\downarrow \\ & & 0 & otherwise \end{bmatrix}$$

The final step for Turing/Church's proof was to construct an algorithm encoding instances (A, D) of the halting problem as statements such that:

$$\Phi_{A,D}\ is\ provable \leftrightarrow A(D)\downarrow$$

Big-Step Semantics:
Ignores intermediate steps and gives result immediately: $E \Downarrow n$
Properties:
Determinacy: Expression evaluation is deterministic (only one result possible): $\forall E, n_1, n_2.\ [E \Downarrow n_1 \wedge E \Downarrow n_2 \Rightarrow n_1 = n_2]$
Totality: Every expression evaluates to something: $\forall E.\ \exists n.\ [E \Downarrow n]$
Example:
Rules: $(\text{B-NUM})\dfrac{}{n \Downarrow n}$ $\quad (\text{B-ADD})\dfrac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = n_1 + n_2$
Derivation: $3 + (2 + 1) \Downarrow 6$

$$(\text{B-ADD})\dfrac{(\text{B-NUM})\dfrac{}{3 \Downarrow 3} \quad (\text{B-ADD})\dfrac{(\text{B-NUM})\dfrac{}{2 \Downarrow 2} \quad (\text{B-NUM})\dfrac{}{1 \Downarrow 1}}{2+1 \Downarrow 3}}{3+(2+1) \Downarrow 6}$$

Small-Step Semantics:
Evaluating an expression step-by-step: $E \rightarrow E'$
Transitive closure $\rightarrow^*$:
$E \rightarrow^* E' \Leftrightarrow E = E' \vee$
$\exists E_1, E_2, \ldots, E_k.\ [E \rightarrow E_1 \rightarrow E_2 \rightarrow \cdots \rightarrow E_k \rightarrow E']$
Example rules: (note + is left-associative)

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

$$(\text{S-LEFT})\dfrac{E_1 \rightarrow E_1'}{E_1 + E_2 \rightarrow E_1' + E_2} \quad (\text{S-RIGHT})\dfrac{E \rightarrow E'}{n + E \rightarrow n + E'}$$

Adding this rule breaks determinacy (maintains confluence):

$$(\text{S-RIGHT-E})\dfrac{E_2 \rightarrow E_2'}{E_1 + E_2 \rightarrow E_1 + E_2'}$$

Normal Form (NF):
E is in it's NF (irreducible) if there is no E' such that $E \rightarrow E'$
Properties:
Determinacy: There is at most one next possible step/rule to apply $\forall E, E_1, E_2.\ [E \rightarrow E_1 \wedge E \rightarrow E_2 \Rightarrow E_1 = E_2]$
Confluence: Determinate → Confluent. Several evaluation paths exist but all get the same end result
$\forall E, E_1, E_2.\ [E \rightarrow^* E_1 \wedge E \rightarrow^* E_2$
$\Rightarrow \exists E'.\ [E_1 \rightarrow^* E' \wedge E_2 \rightarrow *E']]$
Strong Normalisation: All sequences of expressions are finite
Weak N.: There exists a finite sequence of expressions (to normalise) for any expression $\forall E.\ \exists k.\ \exists n.\ [E \rightarrow^k n]$
Unique NF: $\forall E, n_1, n_2.\ [E \rightarrow^* n_1 \wedge E \rightarrow n_2 \Rightarrow n_1 = n_2]$

Specifying Turing Machines: TM to increment a "backwards" binary number:
Define **M = (Q, Σ, s, δ)** where **Σ = {␣, 0, 1}**, **Q = {s, t}**, and **δ**:

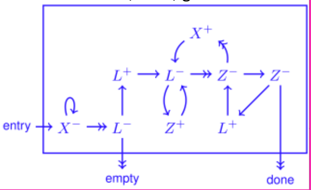
δ	␣	0	1
s	$(t,1,R)$	$(t,1,R)$	$(s,0,R)$
t			

Tracing their execution: remember state is (state, symbols to left of head, symbols on right)
Match state with first symbol on right to get cell for new state. E.g. (s, ε, 1001) $\rightarrow_M$ (s, 0, 001).

States:
A partial function from variables to numbers e.g., $s = (x \mapsto 2, y \mapsto 200, z \mapsto 20)$
Small-step sem $\langle E, s\rangle, \langle B, s\rangle, \langle C, s\rangle$ined using configurations:
While allows statements with side-effects hence $s' = s[x \mapsto u]$ updated after an evaluation
$dom(s') = dom(s)$
$\wedge \forall y.[y \neq x \rightarrow s(y) = s'(y) \wedge s'(x) = a]$

Normalising:
$\rightarrow_e$ and $\rightarrow_b$ are normalising but $\rightarrow_c$ may not be, e.g.,
while $true$ do $skip$
$\langle$while $true$ do $skip, s\rangle$
$\rightarrow_c^3 \langle$while $true$ do $skip, s\rangle$
we have gone through +1 steps to get the same configuration, hence infinite loop

Gadget: Pop L to X. If L = 0, return X = 0 and go to "empty" else if L = $\langle\!\langle x, \ell \rangle\!\rangle$ = n then return X = n, L = ℓ, goto "done".



Properties: $\rightarrow_c, \rightarrow_e$, and $\rightarrow_b$ are deterministic
$\forall C, C_1, C_2 \in Com\forall s, s_1, s_2.$
$[\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle \rightarrow \langle C_1, s_1\rangle = \langle C_2, s_2\rangle]$
and confluent
$\forall C, C_1, C_2 \in Com\forall s, s_1, s_2.$
$[\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle \rightarrow$
$\exists C' \in Com, s'.[\langle C_1, s_1\rangle \rightarrow_c \langle C', s'\rangle \wedge \langle C_2, s_2\rangle \rightarrow_c \langle C', s'\rangle]]$

Configurations:
Answer: expression cannot be simplified further, e.g., the configuration $\langle skip, s\rangle$ as:
$\neg\exists C \in Com, s, s'.$
$[\langle skip, s\rangle \rightarrow_c \langle C, s'\rangle]$
Stuck: when a configuration cannot be evaluated to a NF
$\langle y, (x \mapsto 3)\rangle$
Note a state which reduces to a stuck configuration is not stuck
$\langle 5 < y, (x \mapsto 2)\rangle$

Short-Circuit Semantics:
Note with side effects we can't always short circuit as the RHS can cause side effects

$$\dfrac{}{true\&B \rightarrow_b B} \quad \dfrac{}{false\&B \rightarrow_b false} \quad \dfrac{B_1 \rightarrow_b B_1'}{B_1\&B_2 \rightarrow_b B_1'\&B_2}$$

Strictness:
An operation is strict when arguments must be evaluated before the operation. Addition is strict. Due to short circuiting, & is left strict as the operation can be evaluated without evaluating the right (non-strict in right argument).

$[\ \forall C, s, s'.(\langle C, s\rangle \Downarrow_1 s' \Longrightarrow \langle f(C), s\rangle \Rightarrow s')$
$\wedge$
$\forall k.[\ \forall i[\ i \leq k \Longrightarrow \forall C, s, s'.(\langle C, s\rangle \Downarrow_i s' \Longrightarrow \langle f(C), s\rangle \Rightarrow s')\ ]$
$\Longrightarrow$
$\forall C, s, s'.(\langle C, s\rangle \Downarrow_{k+1} s' \Longrightarrow \langle f(C), s\rangle \Rightarrow s')\ ]\ ]$
$\Longrightarrow$
$\forall i, C, s, s'.(\langle C, s\rangle \Downarrow_i s' \Longrightarrow \langle f(C), s\rangle \Rightarrow s')$

Reminder: Induction over Nat +Tree
$P(\texttt{zero})$
$\wedge$
$\forall N \in Nat.[\ P(N) \Rightarrow P(\texttt{succ}(N))\ ]$
$\Longrightarrow$
$\forall N \in Nat.[\ P(N)\ ]$

Base Case: To Show $\texttt{plus(zero, zero)} = \texttt{zer}$
Proof of base case
$\texttt{plus(zero,zero)} = \texttt{zero}$ (by def. of plus)
Inductive Case: Take $K \in Nat$, arbitrary.
Inductive hypothesis: $\texttt{plus}(K, \texttt{zero}) = K$
Show: $\texttt{plus}(\texttt{succ}(K), \texttt{zero}) = \texttt{succ}(K)$
Proof of inductive case:
$\texttt{plus}(\texttt{succ}(K), \texttt{zero}) = \texttt{succ}(\texttt{plus}(K, \texttt{zero}))$
$= \texttt{succ}(K)$ (by IH)

$\texttt{bTree} \in Tree ::= \texttt{Leaf} \mid \texttt{Node(bTree,bTree)}$
$P(\texttt{Leaf})$
$\wedge$
$\forall \texttt{bt1}, \texttt{bt2} \in Tree.$
$[\ P(\texttt{bt1}) \wedge P(\texttt{bt2}) \Rightarrow P(\texttt{Node(bt1,bt2)})\ ]$
$\Longrightarrow$
$\forall \texttt{bt} \in Tree.P(\texttt{bt})$

Base Case: To Show: $\texttt{leaves(Leaf)} = \texttt{nodes(Leaf)} + 1$.
Proof of Base Case:
$\texttt{leaves(Leaf)}$ = 1 (by def. leaves)
= 0 + 1 by arithmetic
= $\texttt{nodes(Leaf)} + 1$ (by def. nodes)

Inductive Case: Take $bt_1, bt_2 \in$ bTree, arbitrary.
I.H.: $\texttt{leaves}(bt_1) = \texttt{nodes}(bt_1) + 1$
$\wedge$
$\texttt{leaves}(bt_2) = \texttt{nodes}(bt_2) + 1$
To show: $\texttt{leaves}(\texttt{Node}(bt_1, bt_2)) = \texttt{nodes}(\texttt{Node}(bt_1, bt_2)) + 1$
Proof of Inductive Case:
$\texttt{leaves}(\texttt{Node}(bt_1, bt_2))$ =
= $\texttt{leaves}(bt_1) + \texttt{leaves}(bt_2)$ (by def. leaves)
= $\texttt{nodes}(bt_1) + 1 + \texttt{nodes}(bt_2) + 1$ (by I.H.)
= $(\texttt{nodes}(bt_1) + \texttt{nodes}(bt_2) + 1) + 1$ (by arithmetic)
= $\texttt{nodes}(\texttt{Node}(bt_1, bt_2)) + 1$ (by def. nodes)

Boolean Semantics + Induction:
$B \in Bool ::= b \mid B\&B \mid \neg B \qquad b \in \mathbb{B} ::= \texttt{true} \mid \texttt{false}$
Small-Step:

$$\text{Bs1}\ \dfrac{}{\texttt{false}\ \&B_2 \rightarrow \texttt{false}} \quad \text{Bs2}\ \dfrac{}{\texttt{true}\ \&B_2 \rightarrow B_2}$$

$$\text{Bs3}\ \dfrac{B_1 \rightarrow B_1'}{B_1\&B_2 \rightarrow B_1'\&B_2} \quad \text{Bs4}\ \dfrac{B \rightarrow B'}{\neg B \rightarrow \neg B'}$$

$$\text{Bs5}\ \dfrac{}{\neg\texttt{true} \rightarrow \texttt{false}} \quad \text{Bs6}\ \dfrac{}{\neg\texttt{false} \rightarrow \texttt{true}}$$

Big-Step:

$$\text{BL1}\ \dfrac{B_1 \Downarrow \texttt{false}}{B_1\&B_2 \Downarrow \texttt{false}} \quad \text{BL2}\ \dfrac{B_1 \Downarrow \texttt{true} \quad B_2 \Downarrow b}{B_1\&B_2 \Downarrow b}$$

$$\text{BL3}\ \dfrac{B \Downarrow \texttt{true}}{\neg B \Downarrow \texttt{false}} \quad \text{BL4}\ \dfrac{B \Downarrow \texttt{false}}{\neg B \Downarrow \texttt{true}} \quad \text{BL5}\ \dfrac{}{b \Downarrow b}$$

To Prove + Inductive Principle:
$P(\texttt{true})$
$\wedge \qquad \forall B \in Bool.\forall b \in \mathbb{B}.[\ B \Downarrow b \Longrightarrow B \rightarrow^* b\ ]$
$P(\texttt{false})$
$\wedge$
$\forall B_1, B_2 \in Bool.[\ P(B_1) \wedge P(B_2) \rightarrow P(B_1\&B_2)\ ]$
$\wedge$
$\forall B \in Bool.[\ P(B) \Downarrow P(\neg B)\ ]$
$\Longrightarrow$
$\forall B \in Bool.[\ P(B)\ ]$

For *Exp* and *Bool*, we have proofs by induction on the structure of expressions/Booleans. For $\Downarrow_c$ it is more complex as the $\Downarrow_c \Leftarrow \rightarrow_c^*$ direction cannot be proven using totality. Instead, complete/strong induction on the length of $\rightarrow_c^*$ is used.
$\forall E, n \in Exp.\forall s, s' \in State.[\langle E, s\rangle \Downarrow_e \langle n, s'\rangle \Leftrightarrow \langle E, s\rangle \rightarrow_e^* \langle n, s'\rangle]$
$\forall B, b \in Bool.\forall s, s' \in State.[\langle B, s\rangle \Downarrow_b \langle b, s'\rangle \Leftrightarrow \langle B, s\rangle \rightarrow_b^* \langle b, s'\rangle]$
$\forall C \in Com.\forall s, s' \in State.[\langle C, s\rangle \Downarrow_c \langle s'\rangle \Leftrightarrow \langle C, s\rangle \rightarrow_c^* \langle skip, s'\rangle]$

Lambda Calculus:
M ::= x (variable)
λx (abstraction)
MM (application, ((M) M) M)
Bound Variables:
λx . M → x is bound within the scope of M
Free Variables:
λx . M → y is free (unbound)
FreeVars (x) = {x}
FreeVars (λx . M) = FreeVars(M) \ {x}
FreeVars (M N) = FreeVars(M) ∪ FreeVars(N)
Closed Term:
λx y z . x y → λ-term with no free vars
Binding Occurrences:
λx y z . (…) → λ-term's parameters (x, y, z)
Left Associativity:
A B C D ≡ (((A) (B)) (C)) (D)
α-Equivalence:
M $=_\alpha$ N iff N = M by renaming bound variables (free vs must have same name).
Substitution:
M[*new*/*old*] = replace free variable *old* with *new* in M

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

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

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

Semantics:

$$\dfrac{}{(\lambda x\ .\ M)\ N \rightarrow_\beta M[N/x]} \quad \dfrac{M \rightarrow_\beta M'}{\lambda x\ .\ M \rightarrow_\beta \lambda x\ .\ M'} \quad \dfrac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \quad \dfrac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'}$$

$$\dfrac{M =_\alpha M' \quad M' \rightarrow_\beta N' \quad N' =_\alpha N}{M \rightarrow_\beta N}$$

A term of the form λx . N M is called a redex. A λ-term may have several different reductions which form a derivation tree.

Multi-Step Reductions: Steps can be combined using the transitive closure of $\rightarrow_\beta$ under α-conversion:
Reflexivity of α-conversion: $\dfrac{M =_\alpha M'}{M \rightarrow_\beta^* M'}$ Transitivity: $\dfrac{M \rightarrow_\beta M' \quad M' \rightarrow_\beta^* M''}{M \rightarrow_\beta^* M''}$

Confluence: All derivation paths in the tree that reach some NF reach the same NF:
$\forall M, M_1, M_2.\ [M \rightarrow_\beta^* M_1 \wedge M \rightarrow_\beta^* M_2 \Rightarrow \exists M'.[M_1 \rightarrow_\beta^* M' \wedge M_2 \rightarrow_\beta^* M']]$
β Normal Forms: The λ-term contains no redexes (can't be further reduced). If an NF exists it is unique.
is in normal form(M) ≜ ∀N. M $\not\rightarrow_\beta$ N
has a normal form(M) ≜ ∃M' . M $\rightarrow_\beta^*$ M' ∧ is in normal form(M')
β-equivalence: M $=_\beta$ N ⇔ ∃T. [M $\rightarrow_\beta^*$ T ∧ N $\rightarrow_\beta^*$ T]

Reduction Order:
Normal: leftmost outermost redex first, always goes to NF
Call By Name: leftmost outermost first.
Call By Values: leftmost innermost redex first.

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

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

η-equivalence:
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.

$$\dfrac{x \notin FV(M)}{\lambda x\ .\ M\ x =_\eta M} \qquad \dfrac{\forall N.\ M\ N =_{\eta+} M'\ N}{M =_{\eta+} M'}$$

λ-definable:
Partial function **f : Nⁿ ⇀ N** is λ-definable iff there is a closed λ-term M where both hold:
$f(x_1, \ldots, x_n) = y \Leftrightarrow M\ x_1 \ldots x_n =_\beta y$
$f(x_1, \ldots, x_n)\uparrow \Leftrightarrow M\ x_1 \ldots x_n$ has no normal form

Church Numerals:
$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\ldots(f\ x)}_{m \text{ times}}\ldots) \quad \underline{n} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\ldots(f\ x)}_{n \text{ times}}\ldots)$
$\underline{m+n} \triangleq (\lambda m\ .\ \lambda n\ .\ \lambda f\ .\ \lambda x\ .\ m\ f\ (n\ f\ x))\ \underline{m}\ \underline{n}$
$\underline{m \times n} \triangleq (\lambda m\ .\ \lambda n\ .\ \lambda f\ .\ m\ (n\ f))\ \underline{m}\ \underline{n}$
$\underline{m^n} \triangleq (\lambda m\ .\ \lambda n\ .\ n\ m)\ \underline{m}\ \underline{n}$
$\underline{m+1} \triangleq (\lambda m\ .\ \lambda f\ .\ \lambda x\ .\ f\ (m\ f\ x))\ \underline{m}$
if $m = 0$ then x_1 else x_2
$\triangleq (\lambda m\ .\ \lambda x_1\ .\ \lambda x_2\ .\ m\ (\lambda z\ .\ x_2)\ x_1)\ \underline{m}$
$newpair(a,b) \triangleq (\lambda a\ .\ \lambda b\ .\ \lambda s\ .\ s\ a\ b)\ a\ b \equiv (\lambda a\ b\ s\ .\ s\ a\ b)\ a\ b$
$first(p) \triangleq p\ (\lambda x\ .\ \lambda y\ .\ x) \equiv p\ (\lambda x\ y\ .\ x)$
$second(p) \triangleq p\ (\lambda x\ .\ \lambda y\ .\ y) \equiv p\ (\lambda x\ y\ .\ y)$
$pred(n) \triangleq (\lambda n\ .\ \lambda f\ .\ \lambda x\ .\ n\ (\lambda g\ .\ \lambda h\ .\ h\ (g\ f))\ (\lambda u\ .\ x)\ (\lambda u\ .\ u))\ \underline{n}$
$\underline{m-n} \triangleq (\lambda m\ .\ \lambda n\ .\ m\ pred\ n)\ \underline{m}\ \underline{n}$

Transitive closures:
Reminder Given a set A, and a relation $R \subseteq A \times A$, we define $R^+ \subseteq A \times A$, and $R^* \subseteq A \times A$ as
- $R^*(a,a') \triangleq a = a' \vee \exists a'' \in A.[\ R(a,a'') \wedge R^*(a'',a')\ ]$
- $R^+(a,a') \triangleq R(a,a') \vee \exists a'' \in A.[\ R(a,a'') \wedge R^+(a'',a')\ ]$

Defined through rules:

$$\text{R*-A}\ \dfrac{}{R^*(a,a)}\ a \in A \quad \text{R*-B}\ \dfrac{R(a,a'') \quad R^*(a'',a')}{R^*(a,a')}$$

$$\text{R+-A}\ \dfrac{R(a,a')}{R^+(a,a')} \quad \text{R+-B}\ \dfrac{R(a,a'') \quad R^+(a'',a')}{R^+(a,a')}$$

Inductive principles:
$\forall a \in A.Q(a,a) \qquad \wedge$
$\forall a, a', a'' \in A.[\ R(a,a'') \wedge Q(a'',a') \Longrightarrow Q(a,a')\ ]$
$\Longrightarrow$
$\forall a, a' \in A.[\ R^*(a,a') \Longrightarrow Q(a,a')\ ]$

$\forall a, a' \in A.[\ R(a,a') \Longrightarrow Q(a,a')\ ] \qquad \wedge$
$\forall a, a', a'' \in A.[\ R(a,a'') \wedge Q(a'',a') \Longrightarrow Q(a,a')\ ]$
$\Longrightarrow$
$\forall a, a' \in A.[\ R^+(a,a') \Longrightarrow Q(a,a')\ ]$

Combinators: Closed λ-term (no free vars)

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	$\triangleq$	$\lambda g\ .\ (\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x))$			

$Y(f) \triangleq (\lambda x \rightarrow f(x(x)))(\lambda x \rightarrow f(x(x)))$

Register machines:

A RM is specified by:
- Finitely many registers **($R_0, R_1, \dots R_n$)** 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, …,** 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

Configurations:

A configuration has the form **$c = (l, r_0, \dots, r_n)$**, where **l** = current label and r_i = current contents of R_i. The initial configuration **$c_0 = (0, r_0, \dots, r_n)$** starts at label 0 and contains the initial contents of each register.

$R_i = x$ [in configuration c] means **$c = (l, r_0, \dots, r_n)$** with **$r_i = x$**.

Computation:

A computation of an RM is a (finite or infinite) sequence of configurations $c_0, c_1, \dots, c_n$ with each **$c = (l, r_0, \dots, r_n)$** in the sequence determining the next configuration by carrying out the instruction L_l with registers containing $r_0, \dots, r_n$.

Partial Functions:

Register machine computation is deterministic – in any non-halting configuration, the next configuration is uniquely determined by the program. The relation between initial and final register contents defined by a register machine program is a partial function.

Properties:

$(x, y) \in f$ and $(x, y') \in f$ implies $y = y'$

A partial function from a set X to a set Y is total if it satisfies $f(x)\downarrow$ for all $x \in X$

"$f(x) = y$" means $(x, y) \in f$

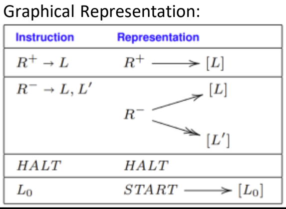
"$f(x)\downarrow$" means $\exists y \in Y\ (f(x) = y)$

"$f(x)\uparrow$" means $\neg\exists y \in Y\ (f(x) = y)$

$X \rightharpoonup Y$ = set of all partial functions from X to Y

$X \to Y$ = set of all (**total**) functions from X to Y

Graphical Representation:



Halting:

Two ways to halt:
1. Jump to instruction L_x where **L_x : HALT**
2. Jump to an undefined instruction ("erroneous halt")

Never halts:
$L_0 : R_1^+ \to L_0$

Example computation from initial configuration (0, 0, 1, 2):

Registers
$R_0\ R_1\ R_2$
Program
$L_0 : R_1^- \to L_1, L_2$
$L_1 : R_0^+ \to L_0$
$L_2 : R_2^- \to L_3, L_4$
$L_3 : R_0^+ \to L_2$
$L_4 : HALT$

Example Computation

L_i	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1
3	2	0	0
2	3	0	0
4	3	0	0

Example function:
$f(x, y) \triangleq x + y$

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

$R_0 = 0, R_1 = x_1, \dots, R_n = x_n$

and all registers set to 0, halts with **$R_0 = y$** iff **$f(x_1, \dots, x_n) = y$**.

Numerical Coding of Pairs:

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

$\langle\langle -, -\rangle\rangle$ 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 $\mathbb{N}$.

Example Decoding Instructions:

$94 = 2^x(2y+1)$
$= 2^1((2\times 23)+1)$
$= \langle\langle 1, 23\rangle\rangle$

$23 = \langle j, k\rangle$
$= 2^j(2k+1) - 1$
$= 24 - 1$
$= 2^3 \times ((2\times 1)+1) - 1$
$= \langle 3, 1\rangle$

$L_0 : R_0^- \to L_3, L_1$

$0 = HALT$, so:
$L_1 : HALT$

$1 = 2^x(2y+1)$
$= 2^0((2\times 0)+1)$
$= \langle\langle 0, 0\rangle\rangle$
$L_2 : R_0^+ \to L_0$

Numerical Coding of Lists:

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

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

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

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

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

Result The function $\ell \mapsto \ulcorner \ell \urcorner$ gives a bijection from $List\ \mathbb{N}$ to $\mathbb{N}$.

The proof follows by observing that

$$\boxed{0b\ulcorner [x_1, x_2, \dots, x_n] \urcorner} = \boxed{1}\boxed{\underbrace{0\cdots0}_{x_n\ 0s}}\boxed{1}\boxed{\underbrace{0\cdots0}_{x_{n-1}\ 0s}}\cdots\boxed{1}\boxed{\underbrace{0\cdots0}_{x_1\ 0s}}$$

To prove $\boxed{0b\ulcorner [x_1, x_2, \dots, x_n] \urcorner} = \boxed{1}\boxed{0\cdots0}\boxed{1}\boxed{0\cdots0}\cdots\boxed{1}\boxed{0\cdots0}$, we use induction on the structure of $L = [x_1, \dots, x_n]$.

Base Case This is trivial as $0b\ulcorner [] \urcorner = 0$.

Inductive step Assume

$$\boxed{0b\ulcorner [x_1, x_2, \dots, x_k] \urcorner} = \boxed{1}\boxed{0\cdots0}\boxed{1}\boxed{0\cdots0}\cdots\boxed{1}\boxed{0\cdots0}$$

By the definitions, we have

$$0b\ulcorner [x, x_1, x_2, \dots, x_k] \urcorner = 0b\langle\langle x, \ulcorner [x_1, \dots, x_k] \urcorner \rangle\rangle = 0b\ulcorner [x_1, \dots, x_n] \urcorner 1 \underbrace{0\ldots0}_{x\ 0s}$$

Numerical Coding of Programs:

If **P** is the RM program

L_0 : body$_0$, L_1 : body$_1$, …, L_n : body$_n$:

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

where the numerical code of the body is:

$$\ulcorner body_0 \urcorner \begin{cases} \ulcorner R_i^+ \to L_j \urcorner \triangleq \langle\langle 2i, j\rangle\rangle \\ \ulcorner R_i^- \to L_j, L_k \urcorner \triangleq \langle\langle 2i+1, \langle j, k\rangle\rangle\rangle \\ \ulcorner HALT \urcorner \triangleq 0 \end{cases}$$

Decoding programs:

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 $R_i^- \to L_j, L_k$

The decoded program **e** is unique, where

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

Example encoding:

$\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner B_0 \urcorner, \dots, \ulcorner B_4 \urcorner] \urcorner$ where

$\ulcorner B_0 \urcorner = \ulcorner R_1^- \to L_1, L_2 \urcorner = \langle\langle (2 \times 1) + 1, \langle 1, 2\rangle\rangle\rangle$
$= \langle\langle 3, 9\rangle\rangle = 8 \times (18 + 1) = 152$

$\ulcorner B_3 \urcorner = \ulcorner R_0^+ \to L_2 \urcorner = \langle\langle 2 \times 0, 2\rangle\rangle = 5$

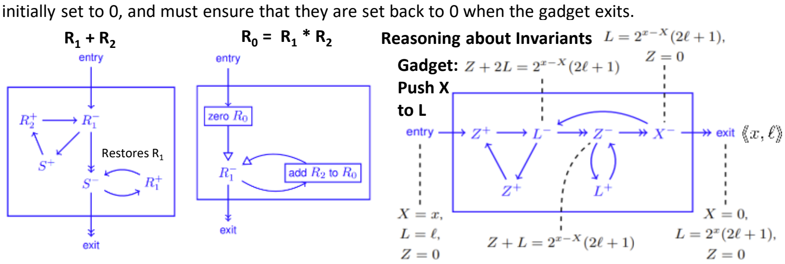
$\ulcorner B_4 \urcorner = \ulcorner HALT \urcorner = 0$

Example decoding:

$\ell : 2^{94} \times 16{,}395 = 2^{94} \times ((2 \times 8{,}197) + 1)$
$= \langle\langle 94, 8{,}197\rangle\rangle$
$\ell = 94 : L_1$ where $\ulcorner \ell_1 \urcorner = 8{,}197$
$\ell_1 : 8{,}197 = 2^x(2y+1)$
$= 2^0((2 \times 4{,}098) + 1)$
$= \langle\langle 0, 4{,}098\rangle\rangle$
$\ell = 94 : 0 : \ell_2$ where $\ulcorner \ell_2 \urcorner = 4{,}098$
$\ell_2 : 4{,}098 = 2^x(2y+1)$
$= 2^1 \times ((2 \times 1{,}024) + 1)$
$= \langle\langle 1, 1{,}024\rangle\rangle$
$\ell = 94 : 0 : 1 : \ell_3$ where $\ulcorner \ell_3 \urcorner = 1{,}024$
$\ell_3 : 1{,}024 = 2^x(2y+1)$
$= 2^{10}((2 \times 0) + 1)$
$= \langle\langle 10, 0\rangle\rangle$
$\ell = 94 : 0 : 1 : 10 : \ell_4$ where $\ulcorner \ell_4 \urcorner = 0$
$\ell_4 : 0 = [\ ]$ $\quad \ell = [94, 0, 1, 10]$

Gadgets:

A gadget is a partial register-machine graph. It has one entry wire, and one or more exit wires. The gadget operates on input and output registers specified in the gadget's name. The gadget may use scratch registers for temporary storage. The gadget assumes the scratch registers are initially set to 0, and must ensure that they are set back to 0 when the gadget exits.

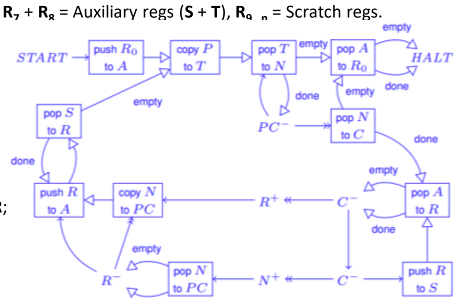


Universal Register Machine:

A register machine which simulates another register machine. It takes arguments **$R_0 = 0$, R_1 = encoded program no. R_2 = encoded arg. list, $R_{3\dots n} = 0$.**

Register usage: **R_1 = Prog**, **R_2 = Args**, **R_3 = PC**, **R_4 = N**ext label/current instr, **R_5 = C**urrent instr, **R_6 = R**eg val for current instr, **$R_7 + R_8$** = Auxiliary regs **(S + T)**, **$R_{9\dots n}$** = Scratch regs.

Steps in URM:
1. Copy **PC^th** item of list in **P** to **N** (halting if **PC** > length of list)
2. If **N** = 0 then halt, else decode **N** as $\langle\langle y, z\rangle\rangle$ **(C** = y, **N** = z)
3. Copy **i^th** item of list in **A** to **R**
4. Execute current instruction on **R**; update **PC** to next label; restore register values to **A**; repeat



Halting Problem for Register Machines:

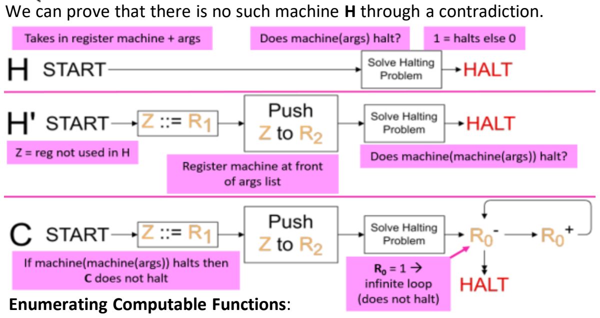
A register machine **H** decides the halting problem if for all **e, a_1, …, a_n N**:

R_0 = 0, R_1 = e, R_2 = $\ulcorner [a_1, \dots, a_n] \urcorner$, $R_{3\dots n}$ = 0

And where **H** halts with the state as follows:

$R_0 \begin{cases} 1 & \text{Register machine encoded as } \mathbf{e} \text{ halts when started with } R_0 = 0, R_1 = a_1, \dots, R_n = a_n \\ 0 & \text{Otherwise} \end{cases}$

We can prove that there is no such machine **H** through a contradiction.



C(C) halts → C with R_1 = ⌜ C ⌝ as an argument does not halt
C(C) does not halt → Then C with R_1 = ⌜C⌝ as an argument **halts**

Both contradictions

Enumerating Computable Functions:

For each **e ∈ N**, let **φ_e ∈ N ⇀ N** be the unary partial function computed by the RM with program **prog(e)**. So, for all **x, y ∈ N**, **$\varphi_e(x) = y$** holds iff the computation of **prog(e)** started with **R_0 = 0, R_1 = x**, and all other registers 0, and eventually halts with **R_0** = y. Thus **e** $\mapsto$ **φ_e** defines an onto function from **N** to the collection of all computable partial functions from **N** to **N**.

Uncomputable Functions:

Let **f ∈ N⇀N** be the partial function **{(x, 0) | $\varphi_x(x)\uparrow$}**. Thus:

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

f is **not computable**, because if it were then **f = φ_e** for some **e ∈ N** hence if $\varphi_e(e)\uparrow$, then f(e) = 0 so $\varphi_e(e)$ = 0, i.e. $\varphi_e(e)\downarrow$
if $\varphi_e(e)\downarrow$, then f(e)↑ so $\varphi_e(e)\uparrow$ **Contradiction!**

Undecidable Sets of Numbers:

Given a subset **S ⊆ N**, its characteristic function **χ_S ∈ N → N** is

$\chi_S(x) \triangleq \begin{cases} 1 \text{ if } x \in S \\ 0 \text{ if } x \notin S \end{cases}$

S ⊆ N is RM decidable if **χ_S** is RM computable.

S_0 ≜ {e | $\varphi_e(0)\downarrow$} is undecidable, because if such an RM exists to solve it, the Halting Problem would be solved. Hence this is a contradiction so the set is undecidable.

S_1 ≜ {e | $\varphi_e(0)$ total function} is undecidable, because if an RM existed to compute **χ_{S1}** we can create another RM checking membership of **S_0** hence undecidable.

Turing Machines:

$$M = (Q, \Sigma, s, \delta) \quad \delta \in (Q \times \Sigma) \rightharpoonup (Q \times \Sigma \times \{L, R\})$$

$\Sigma \triangleq \{\sqcup, 1, 0\}$

TM Def; Finite set of machine states; Initial state **s ∈ Q**; Finite set of tape symbols; Transition function; (symbol at head, current state) ⇀ (write symbol, new state, move L/R)

Transition Function as Table:

	$\sqcup$	1	0
s		$(q_1, 1, L)$	$(q_0, 1, R)$
q_0	$(q_0, 1, L)$	$(q_0, 1, L)$	$(q_0, 1, L)$
q_1	$(q_1, 1, R)$		$(q_1, 0, L)$

No entry = undefined = HALT

Transition:

(q, w, u) →$_M$ (q′, w′, u′)

$\mathbf{first}(u) = (a, u')$

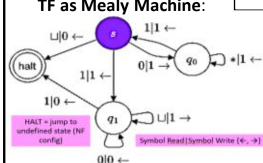
$\delta(q, a) = (q', a', L) \quad \mathbf{last}(w) = (b, w')$

$(q, w, u) \to_M (q', w', ba'u')$

$\mathbf{first}(u) = (a, u') \quad \delta(q, a) = (q', a', R)$

$(q, w, u) \to_M (q', wa', u')$

TF as Mealy Machine:



First and Last: Σ* → Σ x Σ*

These split off the first/last symbols of a string (splitting off ⎵ if the string is empty)

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

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

Configuration:

(q, w, u) where
q = current state **(q ∈ Q)**,
w = finite string of symbols left of head, **u** = right, and **w, u ∈ Σ***

Start State:

(s, ε, u) initial state, no symbols to the left, input to right

Normal Form:

(q, w, u) where **δ(q, a)↑** is undefined for **first(u) = (a, u′)**

Coding a List of Numbers:

A tape over Σ = {⎵, 0, 1} codes a list of numbers if it looks like:

$\underbrace{\cdots}_{\text{all }\sqcup\text{'s}} 0 \underbrace{1\cdots1}_{n_1} \sqcup \underbrace{1\cdots1}_{n_2} \sqcup \cdots \sqcup \underbrace{1\cdots1}_{n_k} 0 \underbrace{\cdots}_{\text{all }\sqcup\text{'s}}$

Which corresponds to the list **[n_1, n_2, …, n_k]**

Note the ⎵s between the 1s

TM = RM:

Theorem: The computation of a TM **M** can be impl by an RM.

Proof: 1. Fix a numerical encoding of **M**'s states, symbols, contents, and configs. **2.** Impl **M**'s transition functions (finite table) using RM instrs on codes. **3.** Impl a RM program to repeatedly carry out →$_M$.

Turing Computability:

f ∈ N^n ⇀ N is TC iff there is TM **M** s.t.:

Starting **M** from init state with head on leftmost 0 of tape coding **[x1, …, xn]**, **M** halts iff **f(x1, . . . , xn)↓**, and the final tape codes a list (length ≥ 1) with first elem **y** where **f(x1, . . . , xn) = y**.