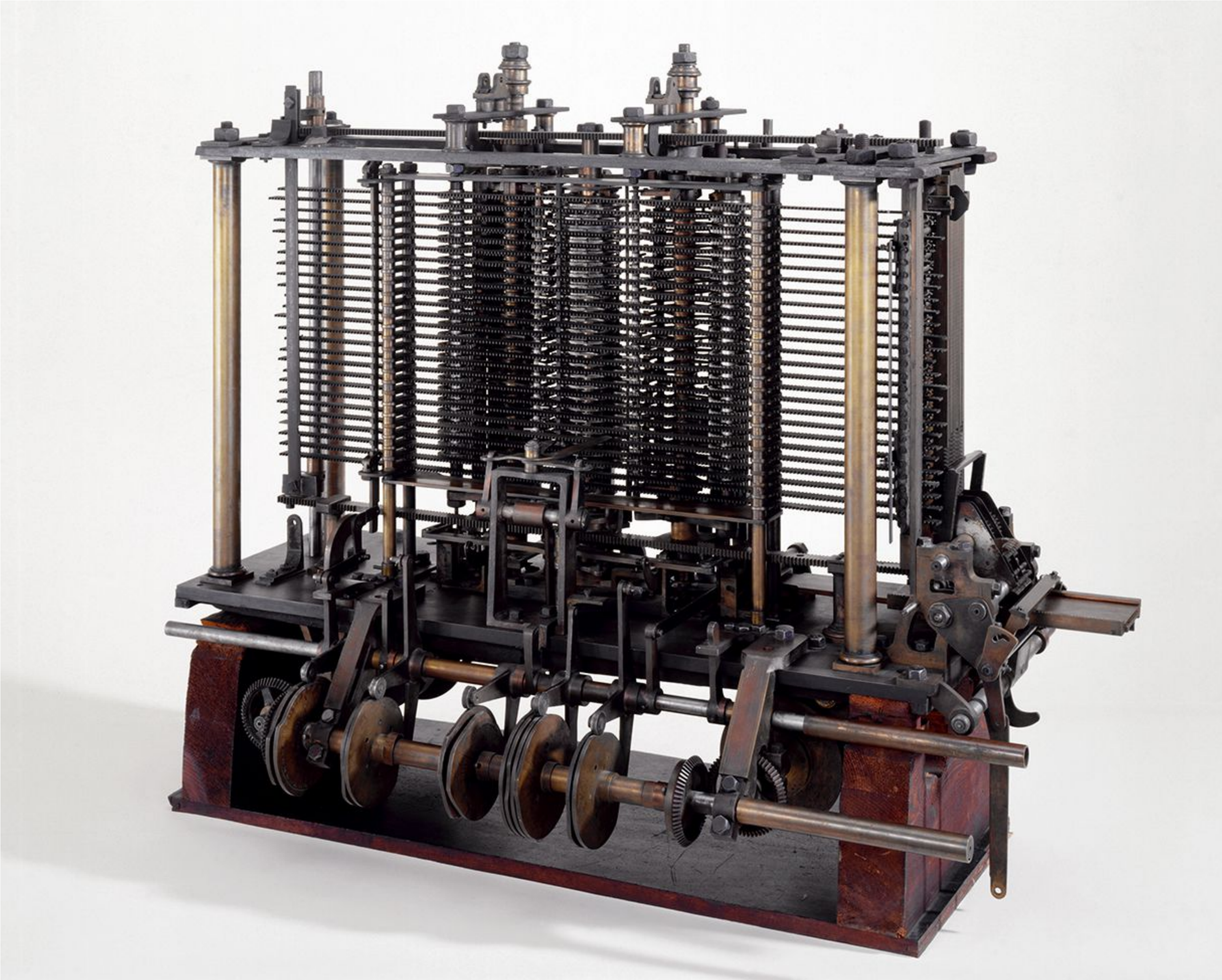


50003

Models of Computation
Imperial College London

Contents

Chapter 1

Introduction

1.1 Course Structure

Dr Azelea Raad

Dr Herbert Wiklicky

First Half

- The while language
- Big & small step semantics
- Structural induction

Second Half

- Register Machines & gadgets
- Turing Machines
- Lambda Calculus

1.2 Algorithms

Euclid's Algorithm — ***Extra Fun!* 1.2.1**

Algorithm to find the greatest common divisor published by greek mathematician Euclid in $\approx$ 300 B.C.

```
-- continually take the modulus and compare until the modulus is zero
euclidGCD :: Int -> Int -> Int
euclidGCD a b
  | b == 0 = a
  | otherwise = euclidGCD b (a `mod` b)
```

Sieve of Eratosthenes — ***Extra Fun!* 1.2.2**

Used to find the prime numbers within a limit. Done by starting from the 2, adding the number to the primes, marking all multiples as non-prime, then repeating progressing to the next non-marked number (a prime) and repeating.

The sieve is attributed to Eratosthenes of Cyrene and was first published $\approx$ 200 B.C.

```
-- Filtering rather than marking elements
eraSieve :: Int -> [Int]
eraSieve lim = eraSieveHelper [2..lim]
  where
    eraSieveHelper :: [Int] -> [Int]
```

```haskell
eraSieveHelper (x:xs) = x:eraSieveHelper (filter (\n -> n `mod` x /= 0) xs)
eraSieveHelper [] = []
```

Al-Khwarizmi ***Extra Fun!* 1.2.3**

A persian polymath who first presented systematic solutions to linear and quadratic equations (by completing the square). He pioneered the treatment of algebra as an independent discipline within mathematics and introduced foundational methods such as the notion of balancing & reducing equal equations (e.g subtract/-cancel the same algebraic term from both sides of an equation)

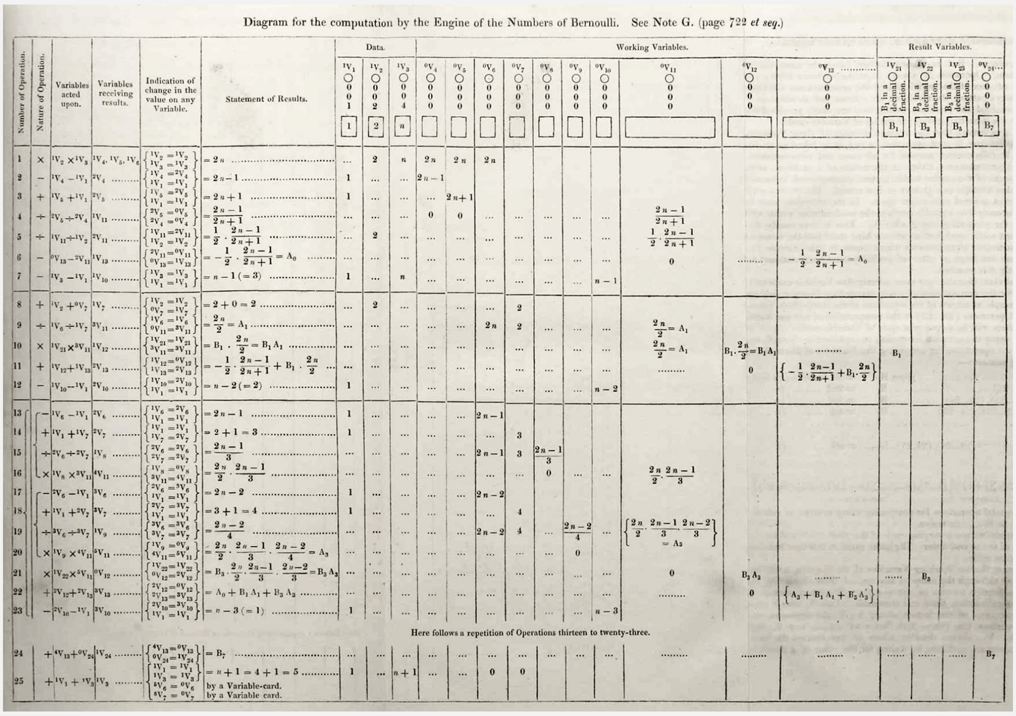
His book title الجبر *"al-jabr"* resulted in the word *algebra* and subsequently algorithm.

Algorithms predate the computer, and have been studied in a mathematical/logical context for centuries.

- Very early attempts such as the Antikythera Mechanism (an analogue calculator for determining the positions of)
- Simple configurable machines (e.g automatic looms, pianola, census tabulating machines) invented in the 1800s.
- Basic calculation devices such as Charles *Babbage's Difference Engine* further generalised the idea of a calculating machine with a sequence of operations, and rudimentary memory store.
- Babbage's Analytical Engine is generally considered the world's first digital computer design, but was not fully implemented due to the limits of precision engineering at the time.
- English mathematician Ada Lovelace writes the first ever computer program (to calculate bernoulli numbers) on Babbage's analytical engine.

Note G ***Extra Fun!* 1.2.4**

While translating a french transcript of a lecture given by Charles Babbage at the University of Turin on his analytical engine, Ada Lovelace added several notes (A-G), with the last including a description of an algorithm to compute the Bernoulli numbers.



Babbage's Machines ***Extra Fun!* 1.2.5**

The *Difference Engine* was used as the basis for designing the fully programmable *Analytical Engine*.

- Held back by lack of funds, limitations of precision machining at the time.
- Contains an ALU for arithmetic operations, supports conditional branches and has a memory

- Part of the machine (including a printing mechanism) are on display at the science museum.

1.3 Decision Problems

Formulas **Definition 1.3.1**

Well formed logical statements that are a sequence of symbols form a given formal language. e.g $(p \vee q) \wedge i$ is a formula, but $) \vee \wedge ji$ is not.

Given:

- A set S of finite data structures of some kind (e.g formulae in first order logic).
- A property P of elements of S (e.g the porperty of a formula that it has a proof).

The associated decision procedure is:

Find an algorithm such that for any $s \in S$, if s has property P the algorithm terminates with 1, otherwise with 0.

1.3.1 Hilbert's Entscheidungsproblem

> *Is there an algorithm which can take any statement in first-order logic, and determine in a finite number of steps if the statement is provable?*

First Order Logic/Predicate Logic **Definition 1.3.2**

An extension of propositional logic that includes quanifiers $(\forall, \exists)$, equality, function symbols (e.g $\times, \div, +, -$) and structured formulas (predicate functions).

This problem was originally presented in a more ambiguous form, using a logic system more powerful than first-order logic.

'*Entscheidungsproblem*' means 'decision problem'

Many tried to solve the problem, without success. One strategy was to try and disprove that such an algorithm can exist. In order to answer this question properly a formal definition of algorithm was required.

1.4 Algorithms

1.4.1 Algorithms Informally

Common features of Algorithms:

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.
Terminate?	Procedure may not terminate on some input data, however we can recognize when it terminates and what the result is.

In 1935/35, Alan Turing (Cambridge) and Church (Princeton) independently gave negative soltuions to Hilberts Entscheidungsproblem (showed such an algorithm could not exist).

1. They gave concrete/precise definitions of what algorithms are (Turing Machines & Lambda Calculus).
2. They regarded algorithms as data, on which other algorithms could act.
3. They reduced the problem to the *Halting problem*.

This work led to the Church-Turing Thesis, that shows everything computable is computed by a Turing Machine. Church's Thesis extended this to show that General Recurisve Functions were the same type as those expressed by lambda calculus, and Turning showed that lambda calculus and the turning machine were equivalent.

Algorithms Formalised

Any formal definition of an algorithm should be:

Precise	No ambiguities, no implicit assumptions, Should be phrased mathematically.
Simple	No unnecessary details, only the few axioms required. Makes it easier to reason about.
General	So all algorithms and types of algorithms are covered.

1.4.2 The Halting Problem

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 $A(D)\downarrow$ holds for $(A, D) \in S$ if algorithm A when applied to D eventually produces a result (halts).

Turning 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:

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

1.4.3 Algorithms as Functions

It is possible to give a mathematical description of a computable function as a special function between special sets.

In the 1960s Strachey & Scott (Oxford) introduced *denotational semantics*, which describes the meaning (denotation) of an algorithm as a function that maps input to output.

Domains **Definition 1.4.1**

Domains are special kinds of partially ordered sets. Partial orders meaning there is an order of elements in the set, but not every element is comparable.

Partial orders are reflexive, transitive and anti-symmetric. You can easily represent them on a Hasse Diagram.

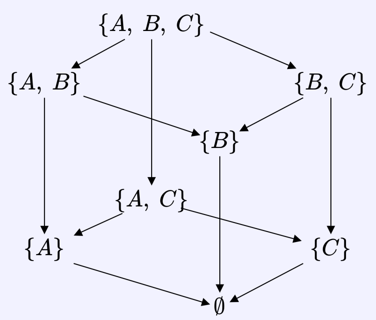


Diagram of $\subseteq$ for sets $\subseteq \{A, B, C\}$

Scott solved the most difficult part, considering recursively defined algorithms as continuous functions between domains.

1.4.4 Haskell Programs

Example using a basic implementation of power.

```haskell
-- Precondition: n >= 0
power :: Integer -> Integer -> Integer
power x 0 = 1
```

```haskell
power x n = x * power x (n-1)

-- Precondition: n >= 0
power' :: Integer -> Integer -> Integer
power' x 0 = 1
power' x n
  | even n = k2
  | odd n  = x * k2
  where
    k  = power' x (n `div` 2)
    k2 = k * k
```

O(n)

power 7 5
$\leadsto$ 7 * (power 7 4)
$\leadsto$ 7 * (7 * (power 7 3))
$\leadsto$ 7 * (7 * (7 * (power 7 2)))
$\leadsto$ 7 * (7 * (7 * (7 * (power 7 1))))
$\leadsto$ 7 * (7 * (7 * (7 * (7 * (power 7 0)))))
$\leadsto$ 7 * (7 * (7 * (7 * (7 * 1))))
$\leadsto$ 16807

O(log(n)) steps

power' 7 5
$\leadsto$ 7 * (power' 7 2)$\hat{2}$
$\leadsto$ 7 * ((power' 7 1)$\hat{2}$)$\hat{2}$
$\leadsto$ 7 * ((7 * (power' 7 0)$\hat{2}$)$\hat{2}$)$\hat{2}$
$\leadsto$ 7 * ((7 * (1)$\hat{2}$)$\hat{2}$)$\hat{2}$
$\leadsto$ 16807

These two functions are equivalent in result however operate differently (one much faster than the other).

1.5 Program Semantics

Denotational Semantics — **Definition 1.5.1**

- A program's meaning is described computationally using denotations (mathematical objects)
- A denotation of a program phrase is built from its sub-phrases.

Operational Semantics — **Definition 1.5.2**

Program's meaning is given in terms of the steps taken to make it run.

There are also *axiomatic semantics* and *declarative semantics* but we will not cover them here.

Chapter 2

While Language

2.1 SimpleExp

We can define a simple expression language (*SimpleExp*) to work on:

$$E \in \mathit{SimpleExp} \ ::= \ n \mid E + E \mid E \times E \mid \ \ldots$$

We want semantics that are the same as we would expect in typical mathematics notation

Small-Set/Structural **Definition 2.1.1**

Gives a method for evaluating an expression step-by-step.

$$E \rightarrow E'$$

Big-Step/Natural **Definition 2.1.2**

Ignores intermediate steps and gives result immediately.

$$E \Downarrow n$$

We need big to define big and small step semantics for SimpleExp to describe this, and have those semantics conform to several properties listed.

2.1.1 Big-Step Semantics

Rules

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

We can similarly define multiplication, subtraction etc.

Properties

Determinacy **Definition 2.1.3**

$$\forall E, n_1, n_2.\ [E \Downarrow n_1 \wedge E \Downarrow n_2 \Rightarrow n_1 = n_2]$$

Expression evaluation is deterministic (only one result possible).

Totality **Definition 2.1.4**

$$\forall E.\ \exists n.\ [E \Downarrow n]$$

Every expression evaluates to something.

Break it! **Example Question 2.1.1**

How could we break the totality of SimpleExp?

$$(\text{B-NON-TOTAL})\frac{}{\mathit{true} \Downarrow \mathit{true}}$$

We can break *totality* by introducing a rule that can always match its output.

The B-NON-TOTAL rule can be applied indefinitely (possible evaluation path that never finishes).

Now it all adds up! **Example Question 2.1.2**

Show that $3 + (2 + 1) \Downarrow 6$ using the provided rules.

We can hence create the derivation:

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

2.1.2 Small Step Semantics

Given a relation $\rightarrow$ we can define a its transitive closure $\rightarrow^*$ such that:

$$E \rightarrow^* E' \Leftrightarrow E = E' \lor \exists E_1, E_2, \dots, E_k. \ [E \rightarrow E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_k \rightarrow E']$$

Rules

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

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

Here we define $+$ as a left-associative operator.

Normal Form **Definition 2.1.5**

E is in its normal form (irreducable) if there is no E' such that $E \rightarrow E'$

In *SimpleExp* the normal form is the natural numbers.

Properties

Confluence **Definition 2.1.6**

$$\forall E, E_1, E_2. \ [E \rightarrow^* E_1 \land E \rightarrow^* E_2 \Rightarrow \exists E'. \ [E_1 \rightarrow^* E' \land E_2 \rightarrow *E']]$$

Determinate $\rightarrow$ Confluent
There are several evaluations paths, but they all get the same end result.

Determinacy **Definition 2.1.7**

$$\forall E, E_1, E_2. \ [E \rightarrow E_1 \land E \rightarrow E_2 \Rightarrow E_1 = E_2]$$

There is at most one next possible step/rule to apply.

Strong Normalisation **Definition 2.1.8**

There are no infinite sequences of expressions, all sequences are finite.

Weak Normalisation **Definition 2.1.9**

$$\forall E. \ \ \exists k. \ \exists n. \ [E \rightarrow^k n]$$

There is some finite sequence of expressions (to normalize) for any expression.

Unique Normal Form **Definition 2.1.10**

$$\forall E, n_1, n_2. \ [E \rightarrow^* n_1 \land E \rightarrow n_2 \Rightarrow n_1 = n_2]$$

To be determined... **Example Question 2.1.3**

Add a rule to break determinacy without breaking confluence.

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

As we can now choose which side to reduce first (S-LEFT or S-RIGHT-E), we have lost determinacy, however we retain confluence.

2.2 While

2.2.1 Syntax

We can define a simple while language (if, else, while loops) to build programs from & to analyse.

$$\begin{array}{rcl} B \in Bool & ::= & true|false|E = E|E < E|B\&B|\neg B \dots \\ E \in Exp & ::= & x|n|E + E|E \times E| \dots \\ C \in Com & ::= & x := E|if\ B\ then\ C\ else\ C|C;C|skip|while\ B\ do\ C \end{array}$$

Where $x \in Var$ ranges over variable identifiers, and $n \in \mathbb{N}$ ranges over natural numbers.

2.2.2 States

Partial Function **Definition 2.2.1**

A mapping of every member of its domain, to at most one member of its codomain.

A *state* is a partial function from variables to numbers (partial function as only defined for some variables). For state s, and variable x, $s(x)$ is defined, e.g:

$$s = (x \mapsto 2, y \mapsto 200, z \mapsto 20)$$

(In the current state, $x = 2, y = 200, z = 20$).

For example:

$$\begin{array}{rcll} s[x \mapsto 7](u) & = & 7 & \text{if } u = x \\ & = & s(u) & \text{otherwise} \end{array}$$

The *small-step* semantics of *While* are defined using *configurations* of form:

$$\langle E, s\rangle, \langle B, s\rangle, \langle C, s\rangle$$

(Evaluating E, B, or C with respect to state s)

We can create a new state, where variable x equals value a, from an existing state s:

$$s'(u) \triangleq \alpha(x) = \begin{cases} a & u = x \\ s(u) & otherwise \end{cases}$$

$$s' = s[x \mapsto u] \text{ is equivalent to } dom(s') = dom(s) \wedge \forall y.[y \neq x \rightarrow s(y) = s'(y) \wedge s'(x) = a]$$

(s' equals s where x maps to a)

2.2.3 Rules

Expressions

$$\text{(W-EXP.LEFT)}\frac{\langle E_1, s\rangle \rightarrow_e \langle E_1', s'\rangle}{\langle E_1 + E_2, s\rangle \rightarrow_e \langle E_1' + E_2, s'\rangle} \qquad \text{(W-EXP.RIGHT)}\frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle n + E, s\rangle \rightarrow_e \langle n + E', s'\rangle}$$

$$\text{(W-EXP.VAR)}\frac{}{\langle x, s\rangle \rightarrow_e \langle n, s\rangle}\ s(x) = n \qquad \text{(W-EXP.ADD)}\frac{}{\langle n_1 + n_2, s\rangle}\ \langle n_3, s\rangle n_3 = n_1 + n_2$$

These rules allow for side effects, despite the While language being side effect free in expression evaluation. We show this by changing state $s \rightarrow_e s'$.

We can show inductively (from the base cases W-EXP.VAR and W-EXP.ADD) that expression evaluation is side effect free.

Booleans

(Based on expressions, one can create the same for booleans) ($b \in \{true, false\}$)

$$\text{(W-BOOL.AND.LEFT)}\frac{\langle B_1, s\rangle \rightarrow_b \langle B_1', s'\rangle}{\langle B_1\&B_2, s\rangle \rightarrow_b \langle B_1'\&B_2, s'\rangle} \qquad \text{(W-BOOL.AND.RIGHT)}\frac{\langle B, s\rangle \rightarrow_b \langle B', s'\rangle}{\langle b\&B_2, s\rangle \rightarrow_b \langle b\&B', s'\rangle}$$

$$\text{(W-BOOL.AND.TRUE)}\frac{}{\langle true\&true, s\rangle \rightarrow_b \langle true, s\rangle} \qquad \text{(W-BOOL.AND.FALSE)}\frac{}{\langle false\&b, s\rangle \rightarrow_b \langle true, s\rangle}$$

(Notice we do not short circuit, as the right arm may change the state. In a side effect free language, we could.)

$$\text{(W-BOOL.EQUAL.LEFT)}\frac{\langle E_1, s\rangle \rightarrow_e \langle E_1', s'\rangle}{\langle E_1 = E_2, s\rangle \rightarrow_b \langle E_1' = E_2, s'\rangle} \qquad \text{(W-BOOL.EQUAL.RIGHT)}\frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle n = E, s\rangle \rightarrow_b \langle n = E, s'\rangle}$$

$$\text{(W-BOOL.EQUAL.TRUE)}\frac{}{\langle n_1 = n_2, s\rangle \rightarrow_b \langle true, s\rangle}\ n_1 = n_2 \qquad \text{(W-BOOL.EQUAL.FALSE)}\frac{}{\langle n_1 = n_2, s\rangle \rightarrow_b \langle false, s\rangle}\ n_1 \neq n_2$$

$$\text{(W-BOOL.LESS.LEFT)}\frac{\langle E_1, s\rangle \rightarrow_e \langle E_1', s'\rangle}{\langle E_1 < E_2, s\rangle \rightarrow_b \langle E_1' < E_2, s'\rangle} \qquad \text{(W-BOOL.LESS.RIGHT)}\frac{\langle E, s\rangle \rightarrow_e \langle E', s'\rangle}{\langle n < E, s\rangle \rightarrow_b \langle n < E, s'\rangle}$$

$$\text{(W-BOOL.LESS.TRUE)}\frac{}{\langle n_1 < n_2, s\rangle \rightarrow_b \langle true, s\rangle}\ n_1 < n_2 \qquad \text{(W-BOOL.EQUAL.FALSE)}\frac{}{\langle n_1 < n_2, s\rangle \rightarrow_b \langle false, s\rangle}\ n_1 \geq n_2$$

$$\text{(W-BOOL.NOT)}\frac{}{\langle \neg true, s\rangle \rightarrow_b \langle false, s\rangle} \qquad \text{(W-BOOL.NOT)}\frac{}{\langle \neg false, s\rangle \rightarrow_b \langle true, s\rangle}$$

Assignment

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

Sequential Composition

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

Conditionals

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

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

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

While

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

2.2.4 Properties

The execution relation ($\rightarrow_c$) is deterministic.

$$\forall C, C_1, C_2 \in Com \forall s, s_1, s_2.[\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle \rightarrow \langle C_1, s_1\rangle = \langle C_2, s_2\rangle]$$

Hence the relation is also confluent:

$$\forall C, C_1, C_2 \in Com \forall s, s_1, s_2.[\langle C, s\rangle \rightarrow_c \langle C_1, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c \langle C_2, s_2\rangle \rightarrow$$
$$\exists C' \in Com, s'.[\langle C_1, s_1\rangle \rightarrow_c \langle C', s'\rangle \wedge \langle C_2, s_2\rangle \rightarrow_c \langle C', s'\rangle]]$$

Both also hold for $\rightarrow_e$ and $\rightarrow_b$.

2.2.5 Configurations

Answer Configuration

A configuration $\langle skip, s\rangle$ is an *answer configuration.* As there is no rule to execute skip, it is a normal form.

$$\neg\exists C \in Com, s, s'.[\langle skip, s\rangle \rightarrow_c \langle C, s'\rangle]$$

For booleans $\langle true, s\rangle$ and $\langle false, s\rangle$ are answer configurations, and for expressions $\langle n, s\rangle$.

Stuck Configurations

A configuration that cannot be evaluated to a normal form is called a *suck configuration.*

$$\langle y, (x \mapsto 3)\rangle$$

Note that a configuration that leads to a *stuck configuration* is not itself stuck.

$$\langle 5 < y, (x \mapsto 2)\rangle$$

(Not stuck, but reduces to a stuck state)

2.2.6 Normalising

The relations $\rightarrow_b$ and $\rightarrow_e$ are normalising, but $\rightarrow_c$ is not as it may not have a normal form.

$$\text{while } \mathit{true} \text{ do } \mathit{skip}$$

$$\langle \text{while } \mathit{true} \text{ do } \mathit{skip}, s\rangle \rightarrow_c^3 \langle \text{while } \mathit{true} \text{ do } \mathit{skip}, s\rangle$$

($\rightarrow_c^3$ means 3 steps, as we have gone through more than one to get the same configuration, it is an infinite loop)

2.2.7 Side Effecting Expressions

If we allow programs such as:

$$\text{do } x := x + 1 \ \mathit{return} \ x$$

$$(\text{do } x := x + 1 \ \mathit{return} \ x) + (\text{do } x := x \times 1 \ \mathit{return} \ x)$$

(value depends on evaluation order)

2.2.8 Short Circuit Semantics

$$\frac{B_1 \rightarrow_b B_1'}{B_1 \& B_2 \rightarrow_b B_1' \& B_2} \qquad \frac{}{\mathit{false} \& B \rightarrow_b \mathit{false}} \qquad \frac{}{\mathit{true} \& B \rightarrow_b B}$$

2.2.9 Strictness

An operation is *strict* when arguments must be evaluated before the operation is evaluated. Addition is struct as both expressions must be evaluated (left, then right).

Due to short circuiting, & is left strict as it is possible for the operation to be evaluated without evaluating the right (*non-strict* in right argument).

2.2.10 Complex Programs

It is now possible to build complex programs to be evaluated with our small step rules.

$$\mathit{Factorial} \triangleq y := x; a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$$

We can evaluate $\mathit{Factorial}$ with an input $s = [x \mapsto \dots]$ to get answer configuration $[\dots, a \mapsto x!, x \mapsto \dots]$

Execute! — **Example Question 2.2.1**

Evaluate $\mathit{Factorial}$ for the following initial configuration:

$$s = [x \mapsto 3, y \mapsto 17, z \mapsto 42]$$

Start

$$\langle y := x; a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), [x \mapsto 3, y \mapsto 17, z \mapsto 42]\rangle$$

Get x variable

where $C = a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 17, z \mapsto 42)$:

$$(\text{W-SEQ.LEFT})\dfrac{(\text{W-ASS.EXP})\dfrac{(\text{W-EXP.VAR})\dfrac{}{\langle x, s\rangle \rightarrow_e \langle 3, s\rangle}}{\langle y := x, s\rangle \rightarrow_c \langle y := 3, s\rangle}}{\langle y := x; C, s\rangle \rightarrow_c \langle y := 3; C, s\rangle}$$

Result:

$$\langle y := 3; a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 17, z \mapsto 42)\rangle$$

Assign to y variable

where $C = a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 17, z \mapsto 42)$:

$$(\text{W-SEQ.LEFT})\dfrac{(\text{W-ASS.NUM})\dfrac{}{\langle y := 3, s\rangle \rightarrow_c \langle \mathit{skip}, s[y \mapsto 3]\rangle}}{\langle y := 3; C, s\rangle \rightarrow_c \langle \mathit{skip}; C, s[y \mapsto 3]\rangle}$$

Result:

$$\langle \mathit{skip}; a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42)\rangle$$

Eliminate skip

where $C = a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42)$:

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

Result:

$$\langle a := 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42)\rangle$$

Assign a

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42)$:

$$\text{(W-SEQ.LEFT)}\frac{\text{(W-ASS.NUM)}\frac{}{\langle a := 1, s\rangle \rightarrow_c \langle skip, s[a \mapsto 1]\rangle}}{\langle a := 1; C, s\rangle \rightarrow_c \langle skip; C, s[a \mapsto 1]\rangle}$$

Result:

$$\langle skip; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Eliminate skip

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$

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

Result:

$$\langle \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Expand while

where $C = (a := a \times y; y := y - 1)$, $B = 0 < y$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

Result:
$\langle \text{if } 0 < y \text{ then } (a := a \times y; y := y - 1; \text{while } 0 < y \text{ do } a := a \times y; y := y - 1) \text{ else } skip, (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$

Get y variable

where $C = (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-COND.BEXP)}\frac{\text{(W-BOOL.LESS.RIGHT)}\frac{\text{(W-EXP.VAR)}\frac{}{\langle y, s\rangle \rightarrow \langle 3, s\rangle}}{\langle 0 < y, s\rangle \rightarrow_b \langle 0 < 3, s\rangle}}{\langle \text{if } 0 < y \text{ then } (C; \text{while } 0 < y \text{ do } C) \text{ else } skip, s\rangle \rightarrow_c \langle \text{if } 0 < 3 \text{ then } (C; \text{while } 0 < y \text{ do } C) \text{ else } skip, s\rangle}$$

Result:
$\langle \text{if } 0 < 3 \text{ then } (a := a \times y; y := y - 1; \text{while } 0 < y \text{ do } a := a \times y; y := y - 1);\ \text{else } skip, (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$

Complete if boolean

where $C = (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-COND.EXP)}\frac{\text{(W-BOOl.LESS.TRUE)}\frac{}{\langle 0 < 3, s\rangle \rightarrow_b \langle true, s\rangle}}{\langle \text{if } 0 < 3 \text{ then } (C; \text{while } 0 < y \text{ do } C) \text{ else } skip, s\rangle \rightarrow_c \langle \text{if } true \text{ then } (C; \text{while } 0 < y \text{ do } C) \text{ else } skip, s\rangle}$$

Result:
$\langle \text{if } true \text{ then } (a := a \times y; y := y - 1; \text{while } 0 < y \text{ do } a := a \times y; y := y - 1);\ \text{else } skip, (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$

Evaluate if

where $C = (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-COND.TRUE)}\frac{}{\langle \text{if } true \text{ then } (C; \text{while } 0 < y \text{ do } C) \text{ else } skip, s\rangle \rightarrow_c \langle C; \text{while } 0 < y \text{ do } C, s\rangle}$$

Result:

$$\langle a := a \times y; y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Evaluate Expression a

where $C = y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-SEQ.LEFT)}\dfrac{\text{(W-ASS.EXP)}\dfrac{\text{(W-EXP.MUL.LEFT)}\dfrac{\text{(W-EXP.VAR)}\dfrac{}{\langle a, s\rangle \rightarrow \langle 1, s\rangle}}{\langle a \times y, s\rangle \rightarrow_e \langle 1 \times y, s\rangle}}{\langle a := a \times y, s\rangle \rightarrow_c \langle a := 1 \times y, s\rangle}}{\langle a := a \times y; C, s\rangle \rightarrow_c \langle a := 1 \times y; C, s\rangle}$$

Result:

$$\langle a := 1 \times y; y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Evaluate Expression y

where $C = y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-SEQ.LEFT)}\dfrac{\text{(W-ASS.EXP)}\dfrac{\text{(W-EXP.MUL.RIGHT)}\dfrac{\text{(W-EXP.VAR)}\dfrac{}{\langle y, s\rangle \rightarrow_e \langle 3, s\rangle}}{\langle 1 \times y, s\rangle \rightarrow_e \langle 1 \times 3, s\rangle}}{\langle a := 1 \times y, s\rangle \rightarrow_c \langle a := 1 \times 3, s\rangle}}{\langle a := 1 \times y; C, s\rangle \rightarrow \langle a := 1 \times 3; C, s\rangle}$$

Result:

$$\langle a := 1 \times 3; y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Evaluate Multiply

where $C = y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-SEQ.LEFT)}\dfrac{\text{(W-ASS.EXP)}\dfrac{\text{(W-EXP.MUL)}\dfrac{}{\langle 1 \times 3, s\rangle \rightarrow_e \langle 3, s\rangle}}{\langle a := 1 \times 3, s\rangle \rightarrow_c \langle a := 3, s\rangle}}{\langle a := 1 \times 3; C, s\rangle \rightarrow_c \langle a := 3; C, s\rangle}$$

Result:

$$\langle a := 3; y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)\rangle$$

Assign 3 to a

where $C = y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

$$\text{(W-SEQ.LEFT)}\dfrac{\text{(W-ASS.NUM)}\dfrac{}{\langle a := 3, s\rangle \rightarrow_c \langle \mathit{skip}, s[a \mapsto 3]\rangle}}{\langle a := 3; C, s\rangle \rightarrow_c \langle \mathit{skip}; C, s[a \mapsto 3]\rangle}$$

Result:

$$\langle \mathit{skip}; y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)\rangle$$

Eliminate Skip

where $C = y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

$$\text{(W-SEQ.SKIP)}\dfrac{}{\langle \mathit{skip}; C, s\rangle \rightarrow_c \langle C, s\rangle}$$

Result:

$$\langle y := y - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)\rangle$$

Assign 3 to y

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

$$\text{(W-SEQ.LEFT)}\dfrac{\text{(W-ASS.EXP)}\dfrac{\text{(W-EXP.SUB.LEFT)}\dfrac{\text{(W-EXP.VAR)}\dfrac{}{\langle y, s\rangle \rightarrow \langle 3, s\rangle}}{\langle y - 1, s\rangle \rightarrow_e \langle 3 - 1, s\rangle}}{\langle y := y - 1, s\rangle \rightarrow_c \langle y := 3 - 1, s\rangle}}{\langle y := y - 1; C, s\rangle \rightarrow_c \langle y := 3 - 1, s\rangle}$$

Result:

$$\langle y := 3 - 1; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)\rangle$$

Evaluate Subtraction

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

$$\text{(W-SEQ.LEFT)}\,\dfrac{\text{(W-ASS.EXP)}\,\dfrac{\text{(W-EXP.SUB)}\,\dfrac{}{\langle 3 - 1, s\rangle \rightarrow_e \langle 2, s\rangle}}{\langle y := 3 - 1, s\rangle \rightarrow_c \langle y := 2, s\rangle}}{\langle y := 3 - 1; C, s\rangle \rightarrow_c \langle y := 2; C, s\rangle}$$

Result:

$$\langle y := 2; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)\rangle$$

Assign 2 to y

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

$$\text{(W-SEQ.LEFT)}\,\dfrac{\text{(W-ASS.NUM)}\,\dfrac{}{\langle y := 2, s\rangle \rightarrow_c \langle \mathit{skip}, s[y \mapsto 2]\rangle}}{\langle y := 2; C, s\rangle \rightarrow_c \langle \mathit{skip}; C, s[y \mapsto 2]\rangle}$$

Result:

$$\langle \mathit{skip}; \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 2, z \mapsto 42, a \mapsto 3)\rangle$$

Eliminate skip

where $C = \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1)$ and $s = (x \mapsto 3, y \mapsto 2, z \mapsto 42, a \mapsto 3)$:

$$\text{(W-SEQ.SKIP)}\,\dfrac{}{\langle \mathit{skip}; C, s\rangle \rightarrow_c \langle C, s\rangle}$$

Result:

$$\langle \text{while } 0 < y \text{ do } (a := a \times y; y := y - 1), (x \mapsto 3, y \mapsto 2, z \mapsto 42, a \mapsto 3)\rangle$$

UNFINISHED!!!

Chapter 3

Structural Induction

3.1 Motivation

Structural induction is used for reasoning about collections of objects, which are:

- structured in a well defined way
- finite but can be arbitrarily large and complex

We can use this is reason about:

- natural numbers
- data structures (lists, trees, etc)
- programs (can be large, but are finite)
- derivations of assertions like $E \Downarrow 4$ (finite trees of axioms and rules)

Structural Induction over Natural Numbers

$$\mathbb{N} \in Nat ::= zero | succ(\mathbb{N})$$

To prove a property $P(\mathbb{N})$ holds, for every number $N \in Nat$ by induction on structure $\mathbb{N}$:

Base Case	Prove $P(zero)$
Inductive Case	Prove $P(Succ(K))$ when $P(K)$ holds

For example, we can prove the property:

$$plus(\mathbb{N}, zero) = \mathbb{N}$$

Base Case

Show $plus(zero, zero) = zero$

(1)	LHS	=	$plus(zero, zero)$	
(2)		=	$zero$	(By definition of $plus$)
(3)		=	RHS	(As Required)

Inductive Case

$N = succ(K)$
Inductive Hypothesis $plus(K, zero) = K$
Show $plus(succ(K), zero) = succ(K)$

(1)	LHS	=	$plus(succ(K), zero)$	
(2)		=	$succ(plus(K, zero))$	(By definition of $plus$)
(3)		=	$succ(K)$	(By Inductive Hypothesis)
(4)		=	RHS	(As Required)

Mathematics induction is a special case of structural induction:

$$P(0) \wedge [\forall k \in \mathbb{N}.P(k) \Rightarrow P(k+1)]$$

In the exam you may use $P(0)$ and $P(K+1)$ rather than $P(zero)$ and $P(succ(k))$ to save time.

3.1.1 Binary Trees

$$bTree \in BinaryTree ::= Node \mid Branch(bTree, bTree)$$

We can define a function *leaves*:

$$\begin{aligned} leaves(Node) &= 1 \\ leaves(Branch(T_1, T_2)) &= leaves(T_1) + leaves(T_2) \end{aligned}$$

Or *branches*:

$$\begin{aligned} branches(Node) &= 0 \\ branches(Branch(T_1, T_2)) &= branches(T_1) + branches(T_2) + 1 \end{aligned}$$

P **Example Question 3.1.1**

rove By induction that $leaves(T) = branches(T) + 1$

UNFINISHED!!!

3.2 Induction over SimpleExp

To define a function on all expressions in *SimpleExp*:

- define $f(n)$ directly, for each number n.
- define $f(E_1 + E_2)$ in terms of $f(E_1)$ and $f(E_2)$.
- define $f(E_1 \times E_2)$ in terms of $f(E_1)$ and $f(E_2)$.

For example, we can do this with *den*:

$$den(E) = n \leftrightarrow E \Downarrow n$$

3.2.1 Many Steps of Evaluation

Given $\rightarrow$ we can define a new relation $\rightarrow^*$ as:

$$E \rightarrow^* E' \leftrightarrow (E = E' \vee E \rightarrow E_1 \rightarrow E_2 \rightarrow \cdots \rightarrow E_k \rightarrow E')$$

For expressions, the final answer is n if $E \rightarrow^* n$.

3.2.2 Multi-Step Reductions

The relation $E \rightarrow^n E'$ is defined using mathematics induction by:

Base Case

$$\forall E \in SImpleExp.\ [E \rightarrow^0 E]$$

Inductive Case

$$\forall E, E' \in SimpleExp.\ [E \rightarrow^{k+1} E' \Leftrightarrow \exists E''.\ [E \rightarrow^k E'' \wedge E'' \rightarrow E']]$$

Definition

$$\forall E, E'.\ [E \rightarrow^* E' \Leftrightarrow \exists n. [E \rightarrow^n E']]$$

$\rightarrow^*$ - there are some number of steps to evaluate to E'

Properties of $\rightarrow$

Determinacy	If $E \rightarrow E_1$ and $E \rightarrow E_2$ then $E_1 = E_2$.
Confluence	If $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$ then there exists E' such that $E_1 \rightarrow^* E'$ and $E_2 \rightarrow^* E'$.
Unique answer	If $E \rightarrow^* n_1$ and $E \rightarrow^* n_2$ then $n_1 = n_2$.
Normal Forms	Normal form is numbers ($\mathbb{N}$) for any E, $E = n$ or $E \rightarrow E'$ for some E'.
Normalisation	No infinite sequences of expressions $E_1, E_2, E_3, \ldots$ such that for all $i \in \mathbb{N}$ $E_1 \rightarrow E_{i+1}$ (Every path goes to a normal form).

3.2.3 Confluence of Small Step

We can prove a lemma expressing confluence:

$L_1 : \forall n \in \mathbb{N}.\forall E, E_1, E_2 \in \mathit{SimpleExp}.[E \rightarrow^n E_1 \wedge E \rightarrow^* E_2 \Rightarrow \exists E' \in \mathit{SimpleExp}.[E_1 \rightarrow^* E' \wedge E_2 \rightarrow^* E']]$

Lemma $\Rightarrow$ Confluence

Confluence is: $\forall E, E_1, E_2 \in \mathit{SimpleExp}.[E \rightarrow^* E_1 \wedge E \rightarrow^* E_2 \Rightarrow \exists E' \in \mathit{SimpleExp}.[E_1 \rightarrow^* E' \wedge E_2 \rightarrow^* E']]$ From lemma L_1

(1)	Take some arbitrary $E, E_1, E_2 \in \mathit{SimpleExp}$, assume confluence holds.	(Initial Setup)
(2)	$E \rightarrow^* E_1$	(By Confluence)
(3)	$\exists n \in \mathbb{N}.[E \rightarrow^n E_1]$	(By 2 & definition of $\rightarrow^*$)
(4)	Hence L_1	(By 3)

3.2.4 Determinacy of Small Step

We create a property P:

$$P(E) \stackrel{def}{=} \forall E_1, E_2 \in \mathit{SimpleExp}.[E \rightarrow E_1 \wedge E \rightarrow E_2 \Rightarrow E_1 = E_2]$$

There are 3 rules that apply:

$$(\mathrm{A})\frac{}{n_1 + n_2 \rightarrow n}\ n = n_1 + n_2 \quad (\mathrm{B})\frac{E \rightarrow E'}{n + E \rightarrow n + E'} \quad (\mathrm{C})\frac{E_1 \rightarrow E_1'}{E_1 + E_2 \rightarrow E_1' + E_2}$$

Base Case

Take arbitrary $n \in \mathbb{N}$ and $E_1, E_2 \in \mathit{SimpleExp}$ such that $n \rightarrow E_1 \wedge n \rightarrow E_2$ to show $E_1 = E_2$.

(1)	$n \not\rightarrow$	(By inversion on A,B & C)
(2)	$\neg(n \rightarrow E_1)$	(By 1)
(3)	$\neg(n \rightarrow E_1 \wedge n \rightarrow E_2)$	(By 2)
(4)	$n \rightarrow E_1 \wedge n \rightarrow E_2 \Rightarrow E_1 = E_2$	(By 3)
(5)	$E \rightarrow E_1 \wedge E \rightarrow E_2 \Rightarrow E_1 = E_2$	(By 4)

Hence $P(n)$

Inductive Step

Take arbitrary E, E_1, E_2 such that $E = E_1 + E_2$
Inductive Hypothesis:

$$IH_1 = P(E_1)$$

$$IH_2 = P(E_2)$$

Assume there exists $E_3, E_4 \in \mathit{SimpleExp}$ such that $E_1 + E_2 \rightarrow E_3$ and $E_1 + E_2 \rightarrow E_4$.
To show $E_3 = E_4$.

From inversion on A, B & C there are 3 cases to consider:
For A:

(1)	There exists $n_1, n_2 \in \mathbb{N}$ such that $E_1 = n_1$ and $E_2 = n_2$	(By case A)
(3)	$E_3 = n_1 + n_2$	(By 1, A)
(4)	$E_4 = n_1 + n_2$	(By 1, A)
(5)	$E_3 = E_4$	(By 3 & 4)

For B:

(1)	There exists $n \in \mathbb{N}$ such that $E_1 = n$	(By case B)
(2)	There exists $E' \in \mathit{SimpleExp}$ such that $E_2 \rightarrow E'$	(By case B)
(3)	$E_3 = n + E'$	(By case B)
(4)	There exists $E'' \in \mathit{SimpleExp}$ such that $E_2 \rightarrow E''$	(By case B)
(5)	$E_4 = n + E''$	(By case B)
(6)	$E' = E''$	(By IH_2)
(7)	$E_3 = E_4$	(By 3,5 & 6)

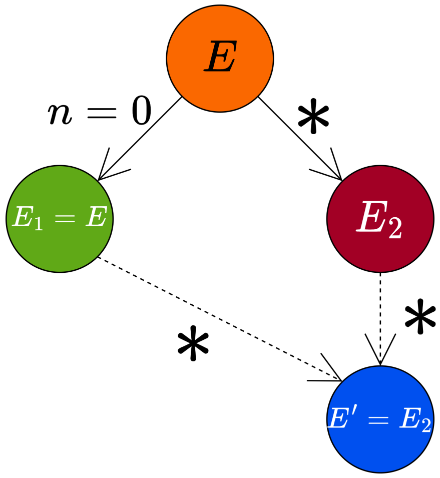
For C:

(1)	There exists $E' \in \mathit{SimpleExp}$ such that $E_1 \to E'$	(By case C)
(2)	There exists $E'' \in \mathit{SimpleExp}$ such that $E_1 \to E''$	(By case C)
(3)	$E_3 = E' + E_2$	(By case C)
(4)	$E_4 = E'' + E_2$	(By case C)
(5)	$E' = E''$	(By IH_1)
(6)	$E_3 = E_4$	(By 3,4 & 5)

(If E reduces to E_1 in n steps, and to E_2 in some number of steps, then there must be some E' that E_1 and E_2 reduce to.)

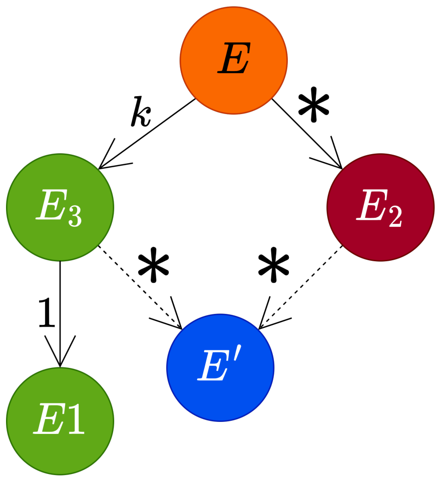
Base Case

The base cases has $n = 0$. Hence $E = E_1$, and hence $E_1 \to^* E_2$ and $E_1 \to^* E'$

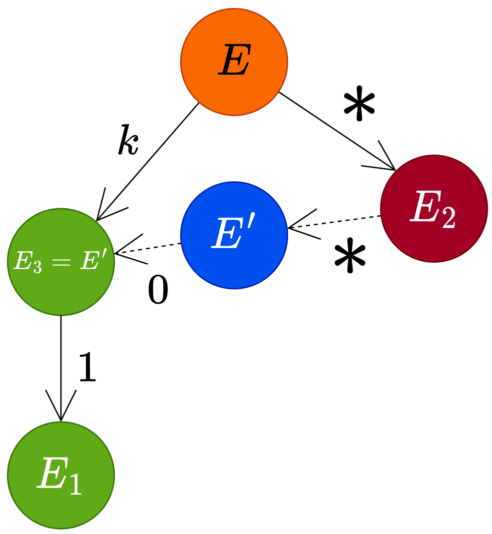


Inductive Case

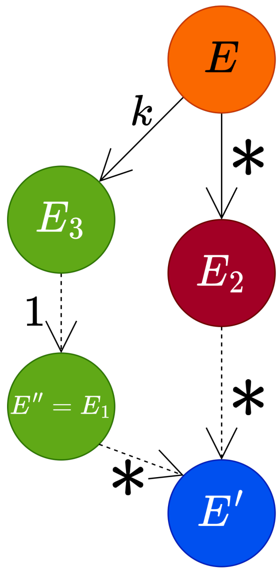
Next we assume confluence for up to k steps, and attempt to prove for $k + 1$ steps.



We have two cases:
Case 1: $E_3 = E'$, this is easy as $E_2 \to^* E' \to^0 E3 \to^1 E1$.



Case 2: $E_3 \rightarrow^1 E'' \rightarrow^* E'$, in this case as $E_3 \rightarrow^1 E1$ we know by determinacy that $E'' = E_1$ and hence $E_1 \rightarrow^* E'$.



3.3 Multi-Step Reductions

Note: We will reference to state by set State $\triangleq (Var \rightharpoonup \mathbb{N})$.

Lemma — **Definition 3.3.1**

A small proven proposition that can be used in a proof. Used to make the proof smaller.

Also know as an "auxiliary theorem" or "helper theorem".

Corollary — **Definition 3.3.2**

A theorem connected by a short proof to another existing theorem.

If B is can be easily deduced from A (or is evident in A's proof) then B is a corollary of A.

3.3.1 Lemmas

1. $\forall r \in \mathbb{N}.\forall E_1, E_1', E_2 \in SimpleExp.[E_1 \rightarrow^r E_1' \Rightarrow (E_1 + E_2) \rightarrow^r (E_1' + E_2)]$
2. $\forall r, n \in \mathbb{N}.\forall E_2, E_2' \in SimpleExp.[E_2 \rightarrow^r E_2' \Rightarrow (n + E_2) \rightarrow^r (n + E_2')]$

3.3.2 Corollaries

1. $\forall n_1 \in \mathbb{N}.\forall E_1, E_2 \in \mathit{SimpleExp}.[E_1 \to^* n_1 \Rightarrow (E_1 + E_2) \to^* (n_1 + E_2)]$
2. $\forall n_1, n_2 \in \mathbb{N}.\forall E_2 \in \mathit{SimpleExp}.[E_2 \to^* n_2 \Rightarrow (n_1 + E_2) \to^* (n_1 + n_2)]$
3. $\forall n, n_1, n_2, \in \mathbb{N}.\forall E_1, E_2 \in \mathit{SimpleExp}.[E_1 \to^* n_1 \wedge E_2 \to^* n_2 \wedge n = n_1 + n_2 \Rightarrow (E_1 + E_2) \to^* n]$

3.3.3 Connecting $\Downarrow$ and $\to^*$ for SimpleExp

$$\forall E \in \mathit{SimpleExp}, n \in \mathbb{N}.[E \Downarrow n \Leftrightarrow E \to^* n]$$

We prove each direction of implication separately. First we prove by induction over E using the property P:

$$P(E) =^{def} \forall n \in \mathbb{N}.[E \Downarrow n \Rightarrow E \to^* n]$$

Base Case

Take arbitrary $m \in \mathbb{N}$ to show $P(m) = m \Downarrow n \Rightarrow m \to^* n$.

(1)	Assume $m \Downarrow n$	
(2)	$m = n$	(From Inversion of $\Downarrow$)
(3)	$m \to^* n$	(By 2 and definition of $\to^*$)

Inductive Step

Take some arbitrary E, E_1, E_2 such that $E = E_1 + E_2$.
Inductive Hypothesis

$$\forall n_1 \in \mathbb{N}.[E_1 \Downarrow n_1 \Rightarrow E_1 \to^* n_1]$$

$$\forall n_2 \in \mathbb{N}.[E_2 \Downarrow n_2 \Rightarrow E_2 \to^* n_2]$$

To show $P(E)$: $\forall n \in \mathbb{N}.[(E_1 + E_2) \Downarrow n \Rightarrow (E_1 + E_2) \to^* n]$.

(1)	Assume $(E_1 + E_2) \Downarrow n$	
(2)	$\exists n_1, n_2 \in \mathbb{N}.[E_1 \Downarrow n_1 \wedge E_2 \Downarrow n_2]$	(By 1 & definition of B-ADD)
(3)	$E_1 \to^* n_1$	(By 2 & IH)
(4)	$E_2 \to^* n_2$	(By 2 & IH)
(5)	Chose some $n \in \mathbb{N}$ such that $n = n_1 + n_2$	
(6)	$(E_1 + E_2) \to^* n$	(By 3,4,5 Corollary 3)
(7)	$E \to^* n$	(By 6, definition of E)

Hence assuming $E \Downarrow n$ implies $E \to^* n$, so $P(E)$.
Next we work the other way, to show:

$$\forall E \in \mathit{SimpleExp}.\forall n \in \mathbb{N}.[E \to^* n \Rightarrow E \Downarrow n]$$

(1)	Take arbitrary $E \in \mathit{SimplExp}$ such that $E \to^* n$	(Initial setup)
(2)	Take some $m \in \mathbb{N}$ such that $E \Downarrow m$	(By totality of $\Downarrow$)
(3)	$n = m$	(By 1,2 & uniqueness of result for $\to$)
(4)	$E \Downarrow n$	(By 3)

It is also possible to prove this without using normalisation and determinacy, by induction on E.

3.3.4 Multi-Step Reductions

Lemmas

$$\forall r \in \mathbb{N}.\forall E_1, E_1', E_2.[E_1 \to^r E_1' \Rightarrow (E_1 + E_2) \to^r (E_1' + E_2)]$$

To prove $\forall r \in \mathbb{N}.[P(r)]$ by induction on r:

Base Case

- Base case is $r = 0$.
- Prove that $P(0)$ holds.

Inductive Step

- Inductive Case is $r = k + 1$ for arbitrary $k \in \mathbb{N}$.
- Inductive hypothesis is $P(k)$.
- Prove $P(k + 1)$ using inductive hypothesis.

Proof of the Lemma

By induction on r: **Base Case:** Take some arbitrary $E_1, E_1', E_2 \in \mathit{SimpleExp}$ such that $E_1 \rightarrow^0 E_1'$.

$$
\begin{array}{lcl}
(1) & E_1 = E_1' & \text{(By definition of } \rightarrow^0\text{)} \\
(2) & (E_1 + E_2) = (E_1' + E_2) & \text{(By 1)} \\
(3) & (E_1 + E_2) \rightarrow^0 (E_1' + E_2) & \text{(By definition of } \rightarrow^0\text{)}
\end{array}
$$

Inductive Step: Take arbitrary $k \in \mathbb{N}$ such that $P(k)$

$$
\begin{array}{lcl}
(1) & \text{Take arbitrary } E_1, E_1', E_2 \text{ such that } E_1 \rightarrow E_1' & \text{(Initial setup)} \\
(2) & \text{Take arbitrary } E_1'' \text{ such that } E_1'' \rightarrow E_1' & \\
(3) & (E_1 + E_2) \rightarrow^k (E_1'' + E_2) & \text{(By 2 \& IH)} \\
(4) & (E_1'' + E_2) \rightarrow (E_1' + E_2) & \text{(By 2 \& rule S-LEFT)} \\
(5) & (E_1 + E_2) \rightarrow^{k+1} (E_1' + E_2) & \text{(3,4, definition of } \rightarrow^{k+1}\text{)}
\end{array}
$$

3.3.5 Determinacy of $\rightarrow$ for Exp

We extend simple expressions configurations of the form $\langle E, s \rangle$.

$$E \in \mathit{Exp} ::= n|x|E + E| \ldots$$

Determinacy:

$$\forall E, E_1, E_2 \in \mathit{Exp}.\forall s, s_1, s_2 \in \mathit{State}.[\langle E, s \rangle \rightarrow \langle E_1, s_1 \rangle \wedge \langle E, s \rangle \rightarrow \langle E_2, s_2 \rangle \Rightarrow \langle E_1, s_1 \rangle = \langle E_2, s_2 \rangle]$$

We prove this using property P:

$$P(E, s) \triangleq \forall E_1, E_2 \in \mathit{Exp}.\forall s_1, s_2 \in \mathit{State}.[\langle E, s \rangle \rightarrow \langle E_1, s_1 \rangle \wedge \langle E, s \rangle \rightarrow \langle E_2, s_2 \rangle \Rightarrow \langle E_1, s_1 \rangle = \langle E_2, s_2 \rangle]$$

Base Case: $E = x$

Take arbitrary $n \in \mathbb{N}$ and $s \in \mathit{State}$ to show $P(n, s)$

$$
\begin{array}{lcl}
(1) & \text{take } E_1 \in \mathit{Exp},\ s_1 \in \mathit{State} \text{ such that } \langle n, s \rangle \rightarrow \langle E_1, s_1 \rangle & \text{(Initial setup)} \\
(2) & \text{take } E_2 \in \mathit{Exp},\ s_2 \in \mathit{State} \text{ such that } \langle n, s \rangle \rightarrow \langle E_2, s_2 \rangle & \text{(Initial setup)} \\
(3) & n = E_1 \wedge s = s_1 & \text{(By 1 \& inversion on definition of E.NUM)} \\
(4) & n = E_2 \wedge s = s_2 & \text{(By 2 \& inversion on definition of E.NUM)} \\
(5) & E_1 = E_2 \wedge s_1 = s_2 & \text{(By 3 \& 4)} \\
(6) & \langle E_1, s_1 \rangle = \langle E_2, s_2 \rangle & \text{(By 5 \& definition of configurations)}
\end{array}
$$

Base Case: $E = x$

Take arbitrary $x \in \mathit{Var}$ and $s \in \mathit{State}$ to show $P(n, s)$

$$
\begin{array}{lcl}
(1) & \text{take } E_1 \in \mathbb{N},\ s_1 \in \mathit{State} \text{ such that } \langle x, s \rangle \rightarrow \langle E_1, s_1 \rangle & \text{(Initial setup)} \\
(2) & \text{take } E_2 \in \mathbb{N},\ s_2 \in \mathit{State} \text{ such that } \langle x, s \rangle \rightarrow \langle E_2, s_2 \rangle & \text{(Initial setup)} \\
(3) & E_1 = s(x) \wedge s_1 = s & \text{(By 1 \& inversion on definition of E.VAR)} \\
(3) & E_2 = s(x) \wedge s_2 = s & \text{(By 2 \& inversion on definition of E.VAR)} \\
(5) & E_1 = E_2 \wedge s_1 = s_2 & \text{(By 3 \& 4)} \\
(6) & \langle E_1, s_1 \rangle = \langle E_2, s_2 \rangle & \text{(By 5 \& definition of configurations)}
\end{array}
$$

... Inductive Step ...

3.3.6 Syntax of Commands

$$C \in \mathit{Com} ::= x := E|\text{if } B \text{ then } C \text{ else } C|C; C|\mathit{skip}|\text{while } B \text{ do } C$$

Determinacy

$$\forall C, C_1, C_2 \in \mathit{Com}.\forall s, s_1, s_2 \in \mathit{State}.[\langle C, s \rangle \rightarrow_c \langle C_1, s_1 \rangle \wedge \langle C, s \rangle \rightarrow_c \langle C_2, s_2 \rangle \Rightarrow \langle C_1, s_1 \rangle = \langle C_2, s_2 \rangle]$$

Confluence

$\forall C, C_1, C_2 \in Com.\forall s, s_1, s_2 \in State.[\langle C, s\rangle \rightarrow_c^* \langle C_1, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c^* \langle C_2, s_2\rangle \Rightarrow \exists C' \in Com.\exists s' \in State.[\langle C_1, s_1\rangle \rightarrow_c^* \langle C', s'\rangle \wedge \langle C_2$

Unique Answer

$$\forall C \in Com.s_1 s_2 \in State.[\langle C, s\rangle \rightarrow_c^* \langle skip, s_1\rangle \wedge \langle C, s\rangle \rightarrow_c^* \langle skip, s_2\rangle \Rightarrow s_1 = s_2]$$

No Normalisation

There exist derivations of infinite length for while.

3.3.7 Connecting $\Downarrow$ and $\rightarrow^*$ for While

1. $\forall E, n \in Exp.\forall s, s' \in State.[\langle E, s\rangle \Downarrow_e \langle n, s'\rangle \Leftrightarrow \langle E, s\rangle \rightarrow_e^* \langle n, s'\rangle]$
2. $\forall B, b \in Bool.\forall s, s' \in State.[\langle B, s\rangle \Downarrow_b \langle b, s'\rangle \Leftrightarrow \langle B, s\rangle \rightarrow_b^* \langle b, s'\rangle]$
3. $\forall C \in Com.\forall s, s' \in State.[\langle C, s\rangle \Downarrow_c \langle s'\rangle \Leftrightarrow \langle C, s\rangle \rightarrow_c^* \langle skip, s'\rangle]$

For *Exp* and *Bool* we have proofs by induction on the structure of expressions/booleans.

For $\Downarrow_c$ it is more complex as the $\Downarrow_c \Leftarrow \rightarrow_c^*$ cannot be proven using totality. Instead *complete/strong induction* on length of $\rightarrow_c^*$ is used.

Chapter 4

Register Machines

4.1 Algorithms

Hilbert's Entscheidungsproblem (Decision Problem) **Definition 4.1.1**

A problem proposed by David Hilbert and Wilhem Ackermann in 1928. Considering if there is an algorithm to determine if any statement is universally valid (valid in every structure satisfying the axioms - facts within the logic system assumed to be true (e.g in maths $1 + 0 = 1$)).

This can be also be expressed as an algorithm that can determine if any first-order logic statement is provable given some axioms.

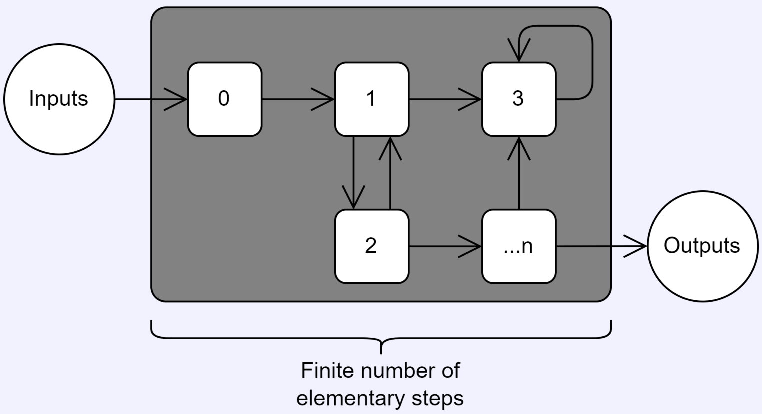
It was proven that no such algorithm exists by Alonzo Church and Alan Turing using their notions of Computing which show it is not computable.

Algorithms Informally **Definition 4.1.2**

One definition is: *A finite, ordered series of steps to solve a problem.*

Common features of the many definitions of algorithms are:

Finite	Finite number of elementary (cannot be broken down further) operations.
Deterministic	Next step uniquely defined by the current.
Terminating?	May not terminate, but we can see when it does & what the result is.



4.2 Register Machines

Register Machine **Definition 4.2.1**

A turing-equivalent (same computational power as a turing machine) abstract machine that models what is computable.

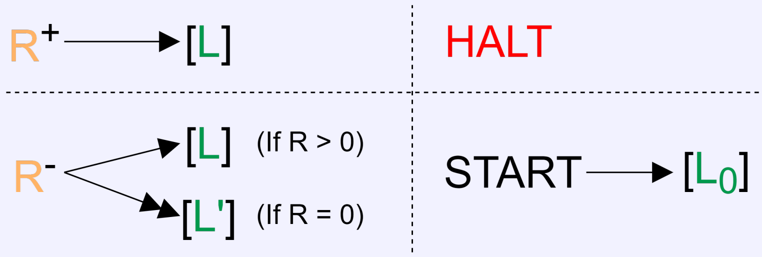
- Infinitely many registers, each storing a natural number ($\mathbb{N} \triangleq \{0, 1, 2, \dots\}$)
- Each instruction has a label associated with it.

There are 3 instructions:

${R_i}^+ \to L_m$	Add 1 to register R_i and then jump to the instruction at L_m
${R_i}^- \to L_n, L_m$	If $R_i > 0$ then decrement it and jump to L_n, else jump to L_m
HALT	Halt the program.

At each point in a program the registers are in a configuration $c = (l, r_0, \dots, r_n)$ (where r_i is the value of R_i and l is the instruction label L_l that is about to be run).

- c_0 is the initial configuration, next configurations are $c_1, c_2, \dots$.
- In a finite computation, the final configuration is the **halting configuration**.
- In a **proper halt** the program ends on a **HALT**.
- In an **erroneous halt** the program jumps to a non-existent instruction, the **halting configuration** is for the instruction immediately before this jump.



Sum of three numbers **Example Question 4.2.1**

The following register machine computes:

$$R_0 = R_0 + R_1 + R_2 \quad R_1 = 0 \quad R_2 = 0$$

Or as a partial function:

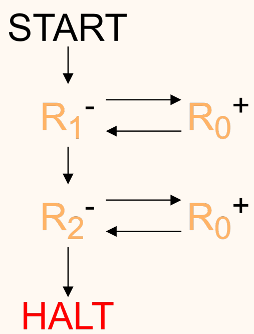
$$f(x, y, z) = x + y + z$$

Registers

$R_0 \quad R_1 \quad R_2$

Program

$L_0 : \quad {R_1}^- \to L_1, L_2$
$L_1 : \quad {R_0}^+ \to L_0$
$L_2 : \quad {R_2}^- \to L_3, L_4$
$L_3 : \quad {R_0}^+ \to L_2$
$L_4 : \quad$ **HALT**



Example Configuration

L_i	R_0	R_1	R_2
0	1	2	3
1	1	1	3
0	2	1	3
1	2	0	3
0	3	0	3
2	3	0	3
3	3	0	2
2	4	0	2
3	4	0	1
2	5	0	1
3	5	0	0
2	6	0	0
4	6	0	0

4.2.1 Partial Functions

Partial Function — **Definition 4.2.2**

Maps some members of the domain X, with each mapped member going to at most one member of the codomain Y.

$$f \subseteq X \times Y \text{ and } (x, y_1) \in f \land (x, y_2) \in f \Rightarrow y_1 = y_2$$

$$\begin{array}{l|l} f(x) = y & (x, y) \in f \\ f(x)\downarrow & \exists y \in Y.[f(x) = y] \\ f(x)\uparrow & \neg\exists y \in Y.[f(x) = y] \\ X \rightharpoonup Y & \text{Set of all } \textit{partial functions} \text{ from } X \text{ to } Y. \\ X \to Y & \text{Set of all } \textit{total functions} \text{ from } X \text{ to } Y. \end{array}$$

A partial function from X to Y is total if it satisfies $f(x)\downarrow$.

Register machines can be considered as partial functions as for a given input/initial configuration, they produce at most one halting configuration (as they are deterministic, for non-finite computations/non-halting there is no halting configuration).

We can consider a register machine as a partial function of the input configuration, to the value of the first register in the halting configuration.

$$f \in \mathbb{N}^n \rightharpoonup \mathbb{N} \text{ and } (r_0, \dots, r_n) \in \mathbb{N}^n, r_0 \in \mathbb{N}$$

Note: Many different register machines may compute the same partial function.

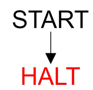
4.2.2 Computable Functions

The following arithmetic functions are computable. Using them we can derive larger register machines for more complex arithmetic (e.g logarithms making use of repeated division).

Projection

$$p(x, y) \triangleq x$$
$$(r_0, r_1) \to r_0$$

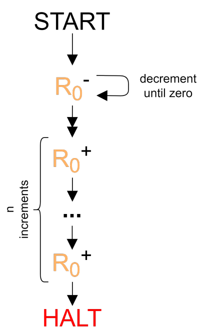
HALT



Constant

$$c(x) \triangleq n$$
$$(r_0) \to n$$

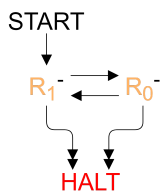
$$\begin{array}{ll} L_0: & {R_0}^- \to L_0, L_1 \\ L_1: & {R_0}^+ \to L_2 \\ \vdots & \vdots \\ L_n: & {R_0}^+ \to L_{n+1} \\ L_{n+1}: & \textbf{HALT} \end{array}$$



Truncated Subtraction

$$x - y \triangleq \begin{cases} x - y & y \leq x \\ 0 & y > x \end{cases}$$
$$(r_0, r_1) \to r_0 - r_1$$

$$\begin{array}{ll} L_0: & {R_1}^- \to L_1, L_2 \\ L_1: & {R_0}^- \to L_0, L_2 \\ L_2: & \textbf{HALT} \end{array}$$

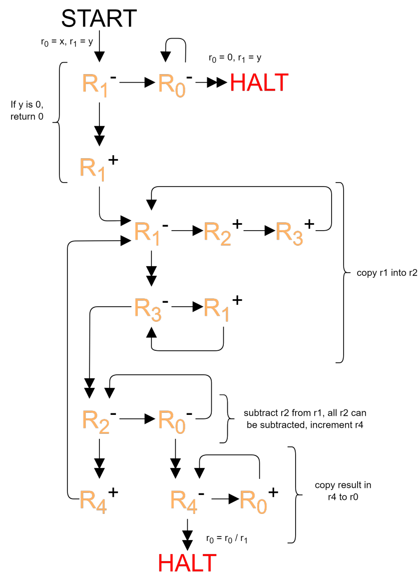


Integer Division

Note that this is an inefficient implementation (to make it easy to follow) we could combine the halts and shortcut the initial zero check (so we don't increment, then re-decrement).

$$x \; div \; y \triangleq \begin{cases} \left\lfloor \dfrac{x}{y} \right\rfloor & y > 0 \\ 0 & y = 0 \end{cases}$$

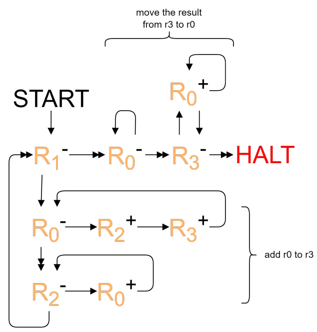
$$\begin{aligned}
L_0 &: \; R_1{}^- \to L_3, L_2 \\
L_1 &: \; R_0{}^- \to L_1, L_2 \\
L_2 &: \; \textbf{HALT} \\
L_3 &: \; R_1{}^+ \to L_4 \\
L_4 &: \; R_1{}^- \to L_5, L_7 \\
L_5 &: \; R_2{}^+ \to L_6 \\
L_6 &: \; R_3{}^+ \to L_4 \\
L_7 &: \; R_3{}^- \to L_8, L_9 \\
L_8 &: \; R_1{}^+ \to L_9 \\
L_9 &: \; R_2{}^- \to L_{10}, L_4 \\
L_{10} &: \; R_0{}^- \to L_9, L_{11} \\
L_{11} &: \; R_4{}^- \to L_{12}, L_{13} \\
L_{12} &: \; R_0{}^+ \to L_{11} \\
L_{13} &: \; \textbf{HALT}
\end{aligned}$$



Multiplication

$$x \times y$$

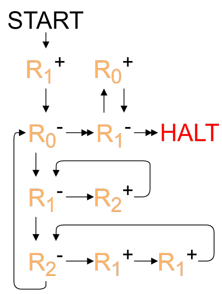
$$\begin{aligned}
L_0 &: \; R_1{}^- \to L_5, L_1 \\
L_1 &: \; R_0{}^- \to L_1, L_2 \\
L_2 &: \; R_3{}^- \to L_3, L_4 \\
L_3 &: \; R_0{}^+ \to L_2 \\
L_4 &: \; \textbf{HALT} \\
L_5 &: \; R_0{}^- \to L_6, L_8 \\
L_6 &: \; R_2{}^+ \to L_7 \\
L_7 &: \; R_3{}^+ \to L_5 \\
L_8 &: \; R_2{}^- \to L_9, L_0 \\
L_9 &: \; R_0{}^+ \to L_8
\end{aligned}$$



Exponent of base 2

$$e(x) \triangleq 2^x$$

$$\begin{aligned}
L_0 &: \; R_1{}^+ \to L_1 \\
L_1 &: \; R_0{}^- \to L_5, L_2 \\
L_2 &: \; R_1{}^- \to L_3, L_4 \\
L_3 &: \; R_0{}^+ \to L_2 \\
L_4 &: \; \textbf{HALT} \\
L_5 &: \; R_1{}^- \to L_6, L_7 \\
L_6 &: \; R_2{}^+ \to L_5 \\
L_7 &: \; R_2{}^- \to L_8, L_1 \\
L_8 &: \; R_1{}^+ \to L_9 \\
L_9 &: \; R_1{}^+ \to L_7
\end{aligned}$$



4.3 Encoding Programs as Numbers

Halting Problem **Definition 4.3.1**

Given a set S of pairs (A, D) where A is an algorithm and D is some input data A operates on $(A(D))$.

We want to create some algorithm H such that:

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

Hence if $A(D) \downarrow$ then $A(D)$ eventually halts with some result.

We can use proof by contradiction to show no such algorithm H can exist.

Assume an algorithm H exists:

$$B(p) \triangleq \begin{cases} halts & H(p(p)) = 0 \quad (p(p) \text{ does not halt}) \\ forever & H(p(p)) = 1 \quad (p(p) \text{ halts}) \end{cases}$$

Hence using H on any $B(p)$ we can determine if $p(p)$ halts $(H(B(p)) \Rightarrow \neg H(p(p)))$.

Now we consider the case when $p = B$.

$B(B)$ **halts**	Hence $B(B)$ does not halt. Contradiction!
$B(B)$ **does not halt**	Hence $B(B)$ halts. Contradiction!

Hence by contradiction there is not such algorithm H.

In order to reason about programs consuming/running programs (as in the halting problem), we need a way to encode programs as data. Register machines use natural numbers as values for input, and hence we need a way to encode any register machine as a natural number.

4.3.1 Pairs

$\langle\langle x, y\rangle\rangle$	$= 2^x(2y+1)$	$y\ 1\ 0_1 \dots 0_x$	Bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}^+ = \{n \in \mathbb{N} \vert n \neq 0\}$
$\langle x, y\rangle$	$= 2^x(2y+1) - 1$	$y\ 0\ 1_1 \dots 1_x$	Bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$

4.3.2 Lists

We can express lists and right-nested pairs.

$$[x_1, x_2, \dots, x_n] = x_1 : x_2 : \dots : x_n = (x_1, (x_2, (\dots, x_n) \dots))$$

We use zero to define the empty list, so must use a bijection that does not map to zero, hence we use the pair mapping $\langle\langle x, y\rangle\rangle$.

$$l : \begin{cases} \ulcorner [] \urcorner \triangleq 0 \\ \ulcorner x_1 :: l_{inner} \urcorner \triangleq \langle\langle x, \ulcorner l_{inner} \urcorner \rangle\rangle \end{cases}$$

Hence:

$$\ulcorner x_1, \dots, x_n \urcorner = \langle\langle x_1, \langle\langle \dots, x_n \rangle\rangle \dots \rangle\rangle$$

4.3.3 Instructions

$$\ulcorner R_i^+ \to L_n \urcorner = \langle\langle 2i, n \rangle\rangle$$

$$\ulcorner R_i^- \to L_n, L_m \urcorner = \langle\langle 2i+1, \langle n, m\rangle \rangle\rangle$$

$$\ulcorner \textbf{HALT} \urcorner = 0$$

4.3.4 Programs

Given some program:

$$\ulcorner \begin{pmatrix} L_0 : & instruction_0 \\ \vdots & \vdots \\ L_n : & instruction_n \end{pmatrix} \urcorner = \ulcorner [\ulcorner instruction_0 \urcorner, \dots, \ulcorner instruction_n \urcorner] \urcorner$$

4.4 Tools

In order to simplify checking workings, a basic python script for running, encoding and decoding register machines is provided (also available in the notes repository).

It is designed to be used in the python shell, to allow for easy manipulation, storing, etc of register machines, encoding/decoding results.

It also produces latex to show step-by-step workings for calculations.

```python
from typing import List, Tuple
from collections import namedtuple

# Register Instructions
Inc = namedtuple('Inc', 'reg label')
Dec = namedtuple('Dec', 'reg label1 label2')
Halt = namedtuple('Halt', '')

'''
 __          ________ _      _____ ____  __  __ ______
 \ \        / /  ____| |    / ____/ __ \|  \/  |  ____|
  \ \  /\  / /| |__  | |   | |   | |  | | \  / | |__
   \ \/  \/ / |  __| | |   | |   | |  | | |\/| |  __|
    \  /\  /  | |____| |___| |___| |__| | |  | | |____
     \/  \/   |______|______\_____\____/|_|  |_|______|

This file can be used to quickly create, run, encode & decode register machine
programs. Furthermore it prints out the workings as formatted latex for easy
use in documents.

Here making use of python's ints as they are arbitrary size (Rust's bigInts
are 3rd party and awful by comparison).

To create register Instructions simply use:
Dec(reg, label 1, label 2)
Inc(reg, label)
Halt()

To ensure your latex will compile, make sure you have commands for, these are
available on my github (Oliver Killane) (Imperial-Computing-Year-2-Notes):

% register machine helper commands:
\newcommand{\instrlabel}[1]{\text{\textcolor{teal}{$L_{#1}$}}}
\newcommand{\reglabel}[1]{\text{\textcolor{orange}{$R_{#1}$}}}
\newcommand{\instr}[2]{\instrlabel{#1} : & $#2$ \\}
\newcommand{\dec}[3]{\reglabel{#1}^- \to \instrlabel{#2}, \instrlabel{#3}}
\newcommand{\inc}[2]{\reglabel{#1}^+ \to \instrlabel{#2}}
\newcommand{\halt}{\text{\textcolor{red}{\textbf{HALT}}}}

To see examples, go to the end of this file.
'''

# for encoding numbers as <a,b>
def encode_large(x: int, y: int) -> int:
    return (2 ** x) * (2 * y+1)

# for decoding n -> <a,b>
def decode_large(n: int) -> Tuple[int, int]:
    x = 0;

    # get zeros from LSB
```

```
    while (n % 2 == 0 and n != 0):
        x += 1
        n /= 2
    y = int((n - 1) // 2)
    return (x,y)

# for encoding <<a,b>> -> n
def encode_small(a: int, b: int) -> int:
    return encode_large(a,b) - 1

# for decoding n -> <<a,b>>
def decode_small(n: int) -> Tuple[int, int]:
    return decode_large(n+1)

# for encoding [a0,a1,a2,...,an] -> <<a0, <<a1, <<a2, <<... <<an, 0 >>...>> >> >> >> -> n
def encode_large_list(lst: List[int]) -> int:
    return encode_large_list_helper(lst, 0)[0]

def encode_large_list_helper(lst: List[int], step: int) -> Tuple[int, int]:
    buffer = r"\to" * step
    if (step == 0):
        print(r"\begin{center}\begin{tabular}{r l l}")
    if len(lst) == 0:
        print(f"{step} &" + rf"$ {buffer} 0$ & (No more numbers in the list, can unwrap recursion) \\")
        return (0, step)
    else:

        print(rf"{step} & $ {buffer} \langle \langle {lst[0]}, \ulcorner {lst[1:]} \urcorner \rangle \ran

        (b, step2) = encode_large_list_helper(lst[1:], step + 1)
        c = encode_large(lst[0], b)

        step2 += 1

        print(f"{step2} & $ {buffer} \langle \langle {lst[0]}, {b} \\rangle \\rangle = {c} $ & (Can now e

        if (step == 0):
            print(r"\end{tabular}\end{center}")
        return (encode_large(lst[0], b), step2)

# decode a list from an integer
def decode_large_list(n : int) -> List[int]:
    return decode_large_list_helper(n, [], 0)

def decode_large_list_helper(n : int, prev : List[int], step : int = 0) -> List[int]:
    if (step == 0):
        print(r"\begin{center}\begin{tabular}{r l l l}")
    if n == 0:
        print(rf"{step} & $0$ & ${prev}$ & (At the list end) \\")
        return prev
    else:
        (a,b) = decode_large(n)
        prev.append(a)
        print(rf"{step} & ${n} = \langle \langle {a}, {b} \rangle \rangle \ \ $&$ {prev}$ & (Decode into

        next = decode_large_list_helper(b, prev, step + 1)

        if (step == 0):
            print(r"\end{tabular}\end{center}")
```

```python
        return next

# For encoding register machine instructions
# R+(i) -> L(j)
def encode_inc(instr: Inc) -> int:
    encode = encode_large(2 * instr.reg, instr.label)
    print(rf"$\ulcorner \inc{{{instr.reg}}}{{{instr.label}}} \urcorner = \langle \langle 2 \times {instr.
    return encode

# R-(i) -> L(j), L(k)
def encode_dec(instr: Dec) -> int:
    encode: int =  encode_large(2 * instr.reg + 1, encode_small(instr.label1 ,instr.label2))
    print(rf"$\ulcorner \dec {{{instr.reg}}}{{{instr.label1}}}{{{instr.label2}}} \urcorner = \langle \lan
    return encode

# Halt
def encode_halt() -> int:
    print(rf"$\ulcorner \halt \urcorner = 0 $")
    return 0

# encode an instruction
def encode_instr(instr) -> int:
    if type(instr) == Inc:
        return encode_inc(instr)
    elif type(instr) == Dec:
        return encode_dec(instr)
    else:
        return encode_halt()

# display register machine instruction in latex format
def instr_to_str(instr) -> str:
    if type(instr) == Inc:
        return rf"\inc{{{instr.reg}}}{{{instr.label}}}"
    elif type(instr) == Dec:
        return rf"\dec{{{instr.reg}}}{{{instr.label1}}}{{{instr.label2}}}"
    else:
        return r"\halt"

# decode an instruction
def decode_instr(x: int) -> int:
    if x == 0:
        return Halt()
    else:
        assert(x > 0)
        (y,z) = decode_large(x)
        if (y % 2 == 0):
            return Inc(int(y / 2), z)
        else:
            (j,k) = decode_small(z)
            return Dec(y // 2, j, k)

# encode a program to a number by encoding instructions, then list
def encode_program_to_list(prog : List) -> List[int]:
    encoded = []
    print(r"\begin{center}\begin{tabular}{r l l}")
    for (step, instr) in enumerate(prog):
        print(f"{step} & ")
        encoded.append(encode_instr(instr))
        print(r"& \\")
    print(r"\end{tabular}\end{center}")
```

```
    print(f"\[{encoded}\]")
    return encoded

# encode a program as an integer
def encode_program_to_int(prog: List) -> int:
    return encode_large_list(encode_program_to_list(prog))

# decode a program by decoding to a list, then decoding each instruction
def decode_program(n : int):
    decoded = decode_large_list(n)
    prog = []
    prog_str = []
    for num in decoded:
        instr = decode_instr(num)
        prog_str.append(instr_to_str(instr))
        prog.append(instr)
    print(f"\[ [ {', '.join(prog_str)} ] \]")
    return prog

# print program in latex form
def program_str(prog) -> str:
    prog_str = []
    for (num, instr) in enumerate(prog):
        prog_str.append(rf"\instr{{{num}}}{{{instr_to_str(instr)}}}")
    print(r"\begin{center}\begin{tabular}{l l}")
    print("\n".join(prog_str))
    print(r"\end{tabular}\end{center}")

# run a register machine with an input:
def program_run(prog, instr_no : int, registers : List[int])-> Tuple[int, List[int]]:
    # step instruction label R0 R1 R2 ... (info)
    print(rf"\begin{{center}}\begin{{tabular}}{{l l l c" + " c" * len(registers) + " }")
    print(r"\textbf{Step} & \textbf{Instruction} & \instrlabel{{i}} &" + " & ".join([rf"$\reglabel{{{n}}}
    print(r"\hline")
    step = 0
    while True:
        step_str = rf"{step} & ${instr_to_str(prog[instr_no])}$ & ${instr_no}$ & " + "&".join([f"${n}$" f
        instr = prog[instr_no]
        if type(instr) == Inc:
            if (instr.reg >= len(registers)):
                print(step_str + rf"(register {instr.reg} is does not exist)\\")
                break
            elif instr.label >= len(prog):
                print(step_str + rf"(label {instr.label} is does not exist)\\")
                break
            else:
                registers[instr.reg] += 1
                instr_no = instr.label
                print(step_str + rf"(Add 1 to register {instr.reg} and jump to instruction {instr.label})
        elif type(instr) == Dec:
            if (instr.reg >= len(registers)):
                print(step_str + rf"(register {instr.reg} is does not exist)\\")
                break
            elif registers[instr.reg] > 0:
                if instr.label1 >= len(prog):
                    print(step_str + rf"(label {instr.label1} is does not exist)\\")
                    break
                else:
                    registers[instr.reg] -= 1
                    instr_no = instr.label1
```

```
                print(step_str + rf"(Subtract 1 from register {instr.reg} and jump to instruction {in
            else:
                if instr.label2 >= len(prog):
                    print(step_str + rf"(label {instr.label2} is does not exist)\\")
                    break
                else:
                    instr_no = instr.label2
                    print(step_str + rf"(Register {instr.reg} is zero, jump to instruction {instr.label2}
        else:
            print(step_str + rf"(Halt!)\\")
            break
        step += 1
    print(r"\end{tabular}\end{center}")
    print("\[(" + ", ".join([str(instr_no)] + list(map(str, registers))) + ")\]")
    return (instr_no, registers)

# Basic tests for program decode and encode
def test():
    prog_a = [
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)]

    prog_b = [
        Dec(1,1,1),
        Halt()
    ]

    # set R0 to 2n for n+3 instructions
    prog_c = [
        Inc(1,1),
        Inc(0,2),
        Inc(0,3),
        Inc(0,4),
        Inc(0,5),
        Inc(0,6),
        Inc(0,7),
        Dec(1, 0, 9),
        Halt()
    ]

    assert decode_program(encode_program_to_int(prog_a)) == prog_a
    assert decode_program(encode_program_to_int(prog_b)) == prog_b
    assert decode_program(encode_program_to_int(prog_c)) == prog_c

# Examples usage
def examples():
    program_run([
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)
    ], 0, [0,7])

    encode_program_to_list([
```

```
        Inc(1,1),
        Inc(0,2),
        Inc(0,3),
        Inc(0,4),
    ])

    encode_program_to_int([
        Dec(1,2,1),
        Halt(),
        Dec(1,3,4),
        Dec(1,5,4),
        Halt(),
        Inc(0,0)
    ])

    decode_program((2 ** 46) * 20483)

# examples()
```

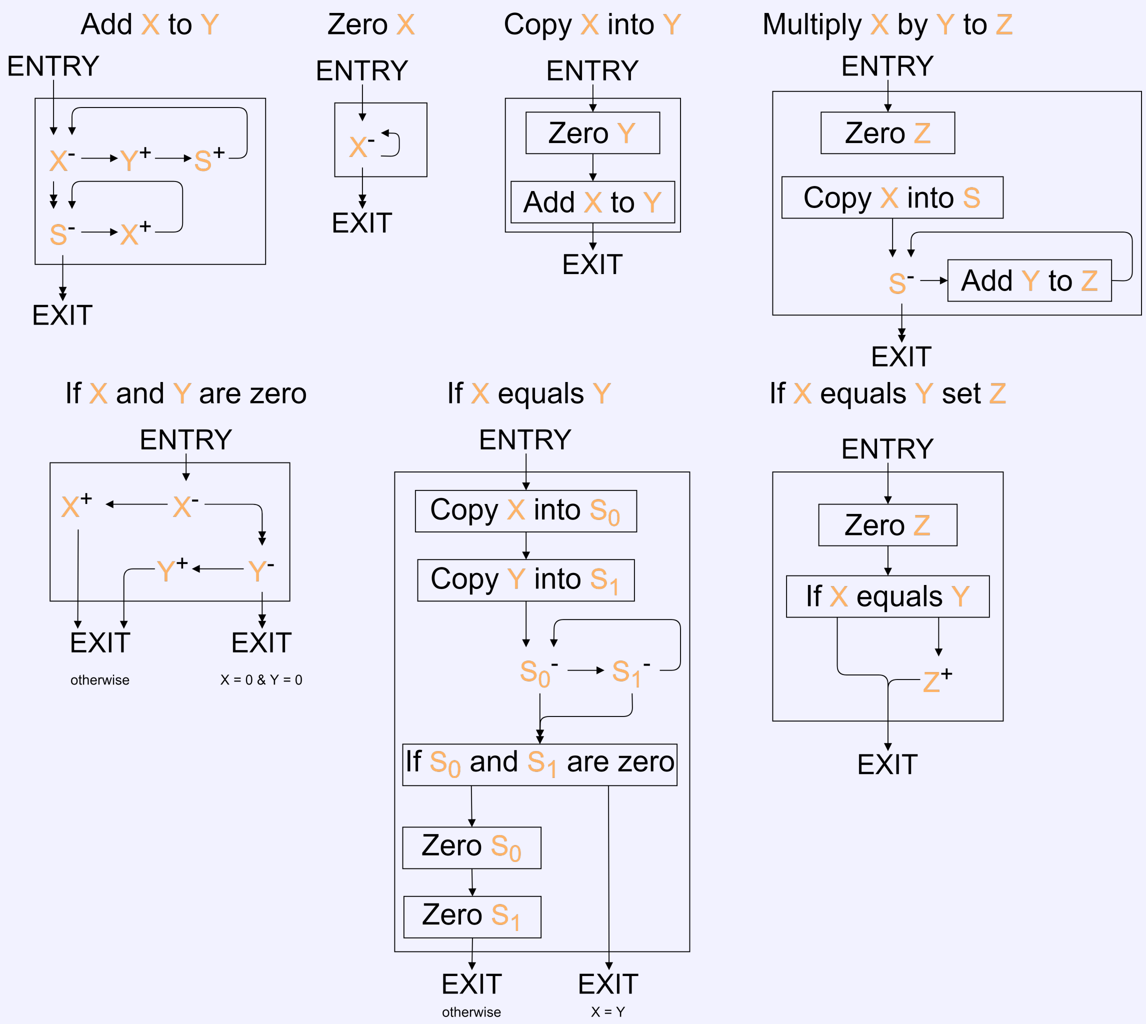
4.5 Gadgets

Register Machine Gadget **Definition 4.5.1**

A gadget is a partial register machine graph, used as components in more complex programs, that can be composed into larger register machines or gadgets.

- Has a single *ENTRY* (much like *START*).
- Can have many *EXIT* (much like **HALT**).
- Operates on registers specified in the name of the gadget (e.g *"Add R_1 to R_2"*).
- Can use scratch registers (assumed to be zero prior to gadget and set to zero by the gadget before it exits - allows usage in loops)
- We can rename the registers used in gadgets (simply change the registers used in the name (*push R_0 to R_1* $\rightarrow$ *push X to Y*), and have all scratch registers renamed to registers unused by other parts of the program)

For example we can create several gadgets in terms of registers that we can rename.



And then can use these to create larger programs.

4.6 Analysing Register Machines

There is no general algorithm for determining the operations of a register machine (i.e halting problem)

However there are several useful strategies one can use:

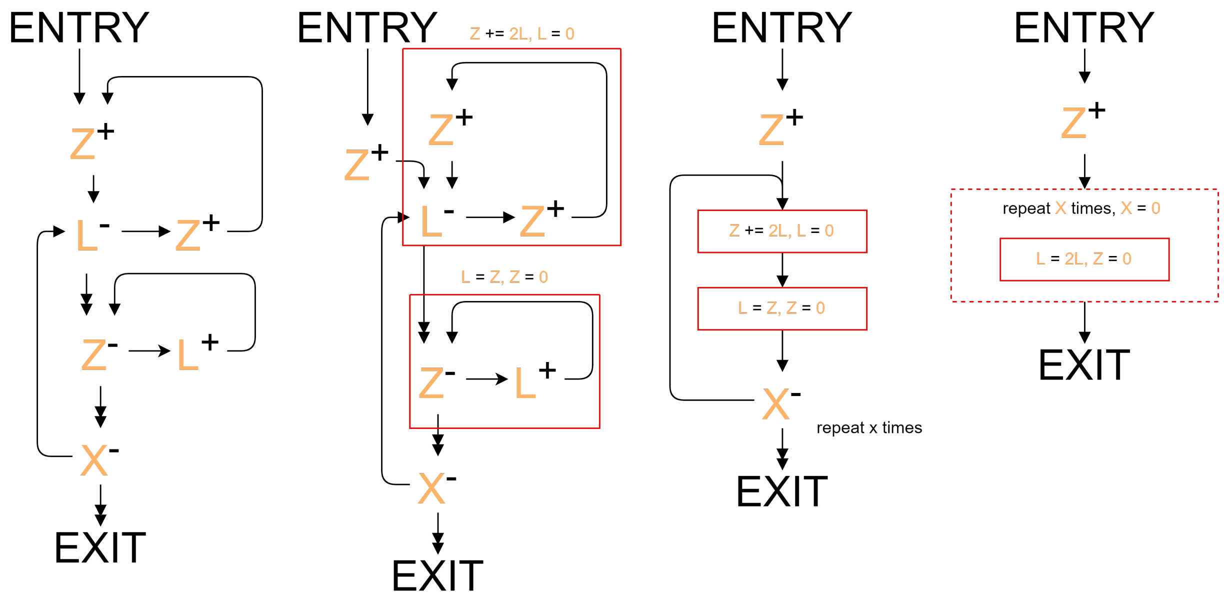
4.6.1 Experimentation

Can create a table of input values against outputs to attempt to fetermine the relation - however the values could match many different relations.

4.6.2 Creating Gadgets

We can group instructions together into gadgets to identify simple behaviours, and continue to merge to develop an understanding of the entire machine.

For example below, we can deduce the result as $L = 2^X(2L + 1)$

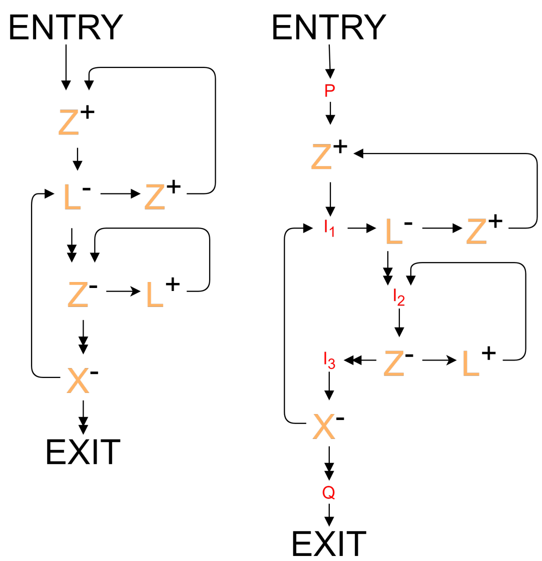


4.6.3 Invariants

We can use logical assertions on the register machine state at certain instructions, both to get the result of the register machine, and to prove the result.

If correct, every execution path to a given instruction's invariant, establishes that invariant.

We could attach invariants to every instruction, however it is usually only necessary to use them at the start, end and for loops (preconditions/postconditions).



Our first invariant (P) can be defined as:

$$P \equiv (X = x \land L = l \land Z = 0)$$

Next we can use the instructions between invariant to find the states under which the invariants must hold.

1.	$P[Z-1/Z] \Rightarrow I_1$	After incrementing Z needs to go to the start of the first loop.
2.	$I_1[L+1/L, Z-2/Z] \Rightarrow I_1$	The loop decrements L and increases Z by two. After each loop iteration, I_1 must still hold.
3.	$I_1 \wedge L = 0 \Rightarrow I_2$	If $L = 0$ the loop is escaped, and we move to I_2.
4.	$I_2[Z+1/Z, L-1/L] \Rightarrow I_2$	Loop increments L and decrements Z on each iteration, after this, I_2 must still hold.
5.	$I_2 \wedge Z = 0 \Rightarrow I_3$	Loop ends when $Z = 0$, moves to I_3.
6.	$I_3[X+1/X] \Rightarrow I_1$	Large loop decrements X on each iteration, invariant must hold with the new/decremented X.
7.	$I_3 \wedge X = 0 \Rightarrow Q$	When the main X-decrementing loop is escaped, we move to exit, so Q must hold.

We can now use these constraints (also called *verification conditions*) to determine an invariant.

For each constraint we do:

1. Get the basic for (potentially one already derived) for the invariant in question.
2. If there is iteration, iterate to build up a disjunction.
3. Find the pattern, and then re-form the invariant based on it.

Constraint 1.

Hence we can deduce I_1 as:

$$I_1 = (X = x \wedge L = l \wedge Z = 1)$$

(Take P and increment Z)

Constraint 2.

We can iterate to get the disjunction:

$$I_1 \equiv (X = x \wedge L = l \wedge Z = 1) \vee (X = x \wedge L + 1 = l \wedge Z - 2 = 1) \vee (X = x \wedge L + 2 = l \wedge Z - 4 = 1) \vee \dots$$

Hence we can determine the pattern for each disjunct as:

$$Z + 2L = 2l + 1$$

Hence we create our invariant:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1)$$

Constraint 3.

Hence as $L = 0$ we can determine that $Z = 2l + 1$.

$$I_2 = (X = x \wedge Z = 2l + 1 \wedge L = 0)$$

Constraint 4.

We iterate to get the disjunction:

$$I_2 = (X = x \wedge Z = 2l + 1 \wedge L = 0) \vee (X = x \wedge Z = 2l + 0 \wedge L = 1) \vee (X = x \wedge Z = 2l - 1 \wedge L = 2) \vee \dots$$

Hence we notice the pattern:

$$Z + L = 2l + 1$$

So can deduce the invariant:

$$I_2 = (X = x \wedge Z + L = 2l + 1)$$

Constraint 5.

We can derive an invariant I_3 using $Z = 0$.

$$I_3 = (X = x \wedge L = 2l + 1 \wedge Z = 0)$$

Constraint 6.

We can use the constraint, and the currently derived I_1 to get a disjunction:

$$I_1 = (X = x - 1 \wedge L = 2l + 1 \wedge Z = 0) \vee (X = x \wedge Z + 2L = 2l + 1)$$

We can apply constraint **2.** on the first part of this disjunction, iterating to get the disjunction:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1) \vee \begin{pmatrix} (X = x - 1 \wedge L = 2l + 1 \wedge Z = 0) \vee \\ (X = x - 1 \wedge L = 2l + 0 \wedge Z = 2) \vee \\ (X = x - 1 \wedge L = 2l - 1 \wedge Z = 4) \vee \\ (X = x - 1 \wedge L = 2l - 2 \wedge Z = 8) \vee \dots \end{pmatrix}$$

Hence for the second group of disjuncts we have the relation:

$$Z + 2L = 2(2l + 1)$$

Hence we have:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1) \vee (X = x - 1 \wedge Z + 2L = 2(2l + 1))$$

Hence when we repeat on the larger loop, we will double again, iterating we get:

$$I_1 = (X = x \wedge Z + 2L = 2l + 1) \vee (X = x - 1 \wedge Z + 2L = 2(2l + 1)) \vee (X = x - 2 \wedge Z + 2L = 4(2l + 1)) \vee \dots$$

Hence we have the relation:

$$I_1 = (Z + 2L = 2^{X-x}(2l + 1))$$

We can apply this doubling to L_2 also as it forms part of the larger loop:

$$I_2 = (Z + L = 2^{X-x}(2l + 1))$$

And to I_3:

$$I_3 = (L = 2^{X-x}(2l + 1) \wedge Z = 0)$$

Constraint 7.

Hence we can now derive Q as:

$$Q = (L = 2^x(2l + 1) \wedge Z = 0)$$

Termination

We also need to show that each of our loops eventually terminate, we can do this by showing that sme variant (e.g register, or combination of) decreases every time the invariant is reached/visited.

For I_1 we can use the lexicographical ordering (X, L) as in each inner loop L decreases, but for the larger loop while L is reset/does not increase, X does.

For I_2 we can similarly use the lexicographical ordering (X, Z)

For I_3 we can just use X.

4.7 Universal Register Machine

A register machine that simulates a register machine.

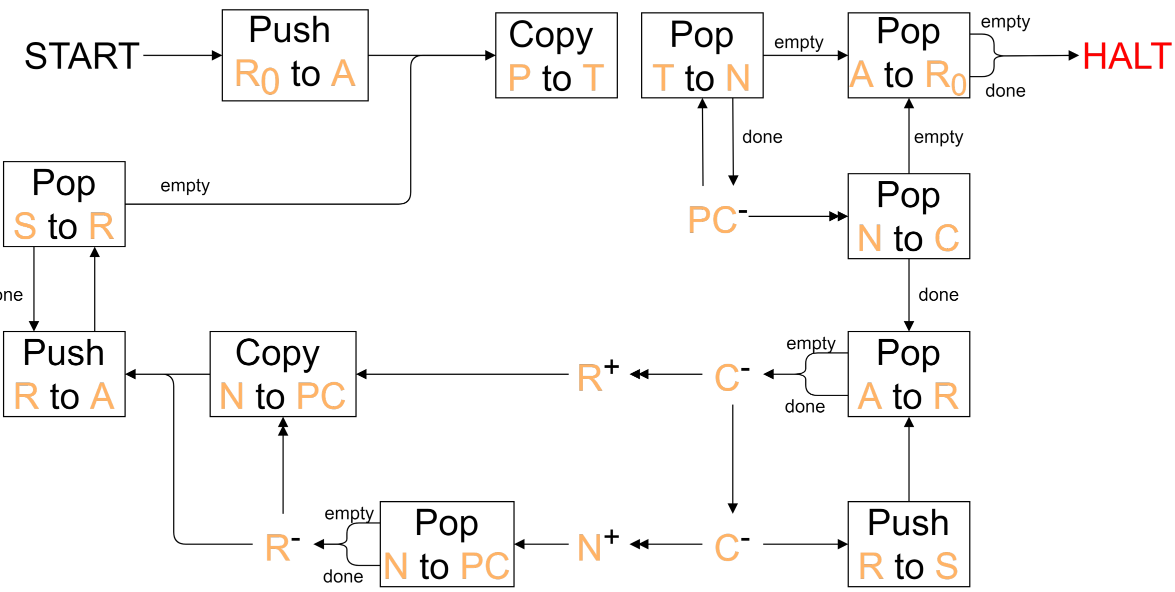
It takes the arguments:

R_0 = 0
R_1 = the program encoded as a number
R_2 = the argument list encoded as a number
All other registers zeroed

The registers used are:

R_1	P	Program code of the register machine being simulated/emulated.
R_2	A	Arguments provided to the simulated register machine.
R_3	PC	Program Counter - The current register machine instruction.
R_4	N	Next label num,ber/next instruction to go to. Is also used to store the current instruction
R_5	C	The current instruction.
R_6	R	The value of the register used by the current instruction.
R_7	S	Auxiliary Register
R_8	T	Auxiliary Register
$R_9 \ldots$		Scratch Registers

```
while true:
    if PC >=  length P:
        HALT!

    N = P[PC]

    if N == 0:
        HALT!

    (curr, next) = decode(N)
    C = curr
    N = next

    # either C = 2i (R+) or C = 2i + 1 (R-)
    R = A[C // 2]

    # Execute C on data R, get next label and write back to registers
    (PC, R_new) = Execute(C, R)
    A[C//2] = R_new
```



Chapter 5

Credit

Image Credit

Front Cover   Analytical Engine - Science Museum London

Content

Based on the *Models of Computation* course taught by Dr Azelea Raad and Dr Herbert Wiklicky.

These notes were written by Oliver Killane.