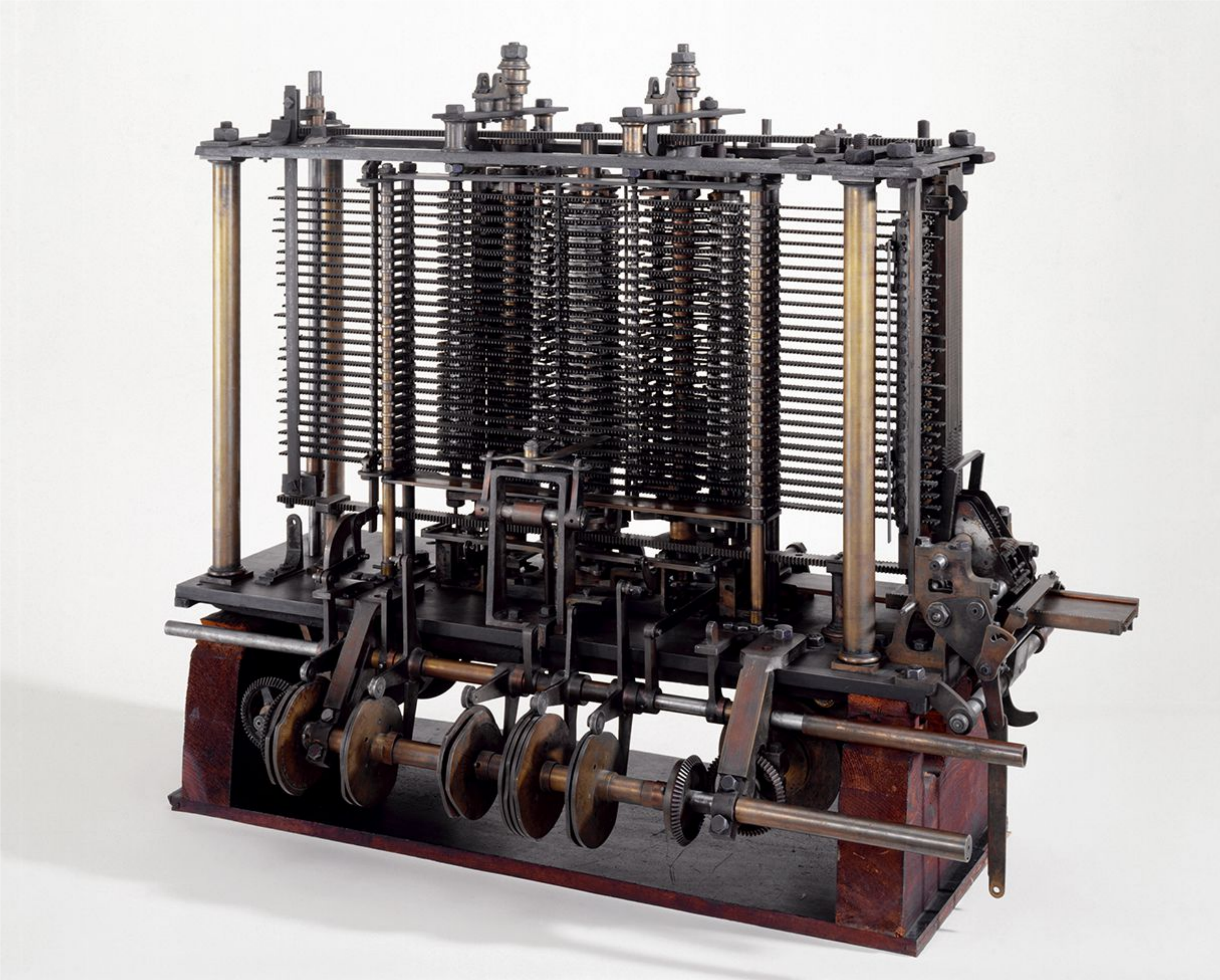


50003

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.

```haskell
-- 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.

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

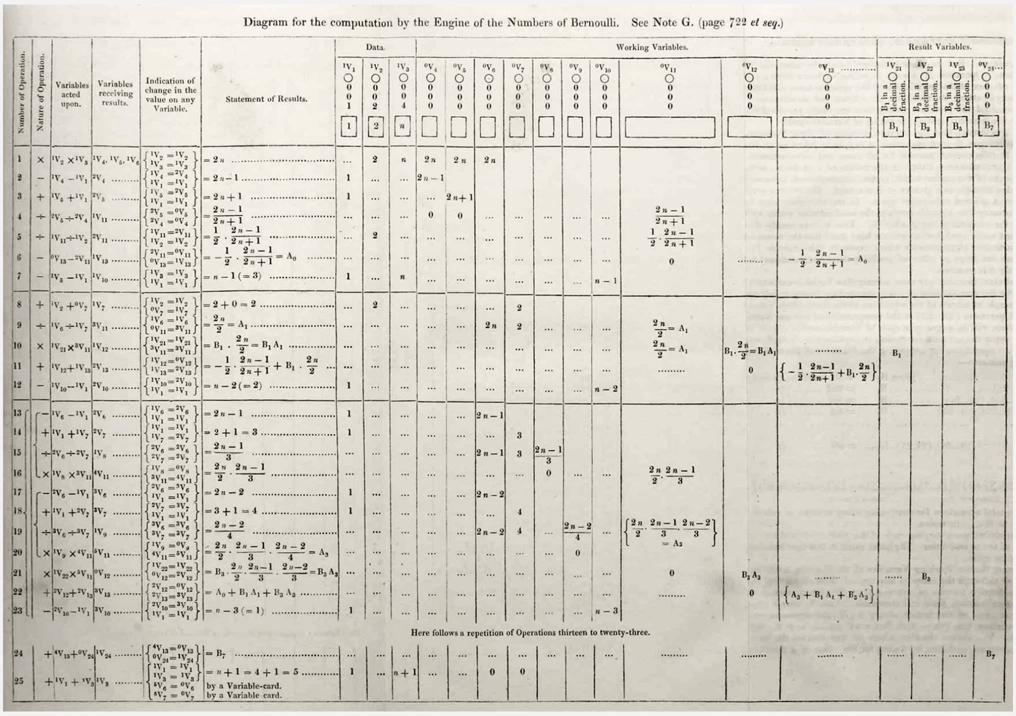
His book title الجبر *"al-jabr"* resulted in the word *algebra* and subsequently algorithm.

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 astronomical positions) $\approx$ 100 B.C.
- 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.



This is the known example of a computer program.

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 $(p \lor q) \land i$ is a formula, but $) \lor \land ji$ 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 property 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:

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

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.

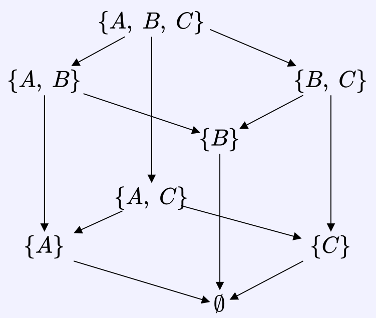


Diagram of $\subseteq$ for sets $\subseteq \{A, B, C\}$

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.

```
-- Precondition: n >= 0
power :: Integer -> Integer -> Integer
power x 0 = 1
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
⇝ 7 * (power 7 4)
⇝ 7 * (7 * (power 7 3))
⇝ 7 * (7 * (7 * (power 7 2)))
⇝ 7 * (7 * (7 * (7 * (power 7 1))))
⇝ 7 * (7 * (7 * (7 * (7 * (power 7 0)))))
⇝ 7 * (7 * (7 * (7 * (7 * 1))))
⇝ 16807

O(log(n)) steps

power’ 7 5
⇝ 7 * (power’ 7 2)2̂
⇝ 7 * ((power’ 7 1)2̂)2̂
⇝ 7 * ((7 * (power’ 7 0)2̂)2̂)2̂
⇝ 7 * ((7 * (1)2̂)2̂)2̂
⇝ 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.

60007 - The Theory and Practice of Concurrent Programming ***Extra Fun!* 1.5.1**

The third-year concurrency module uses both operational and denotational semantics to reason about the correctness of concurrent programs, and possible executions under different memory models (see notes here).

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:

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

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.

$$E \to E'$$

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

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

We can similarly define multiplication, subtraction etc.

Properties

Determinacy — **Definition 2.1.3**

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

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**

We introduce a subtraction operator with big step rule:

$$\text{(B-SUB)}\frac{E_1 \Downarrow n_1 \qquad E_2 \Downarrow n_2}{E_1 - E_2 \Downarrow n_3} \; n_3 = n_1 - n_2$$

What properties of simpleExp does this break? How could this be resolved?

It breaks totality as we specify $n \in \mathbb{N}$, hence $n \geq 0$.

$$\text{for example} \quad (?)\frac{\text{(B-NUM)}\frac{}{3 \Downarrow 3} \qquad \text{(B-NUM)}\frac{}{4 \Downarrow 4}}{3 - 4 \Downarrow ?}$$

We could fix this by:

- Changing the set of n to include negative numbers

- Use saturating arithmetic, and fix negative subtraction to zero by modifying the B-SUB rule to have $n_3 = n_1 - n_2$ where $n_1 \geq n_2$, and introducing a new saturated arithmetic rule for $n_1 < n_2$.
- Add a new result value to represent a non-number/underflow. $n \in \mathbb{N} \cup \{Nan\}$ and set negative results to NaN

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:

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

C Semantics & Short Circuiting in Big-Step — **Example Question 2.1.3**

In this module short-circuiting and side-effects have been kept separate, however this typically not the case (expressions with assignment, using results of functions in expressions).

```
int main() {
        bool a = false;
        bool b = true || (a = true);
        // a is false, b is true
}
```

Create basic big-step operational semantics rules for an extension to SimpleExp boolean expressions that contains:

- Assignments in expressions $B ::= x \mid B \vee B \mid B \wedge B \mid \neg B \mid x := B$ where x is a variable identifier $x \in Var$, assignment evaluates to the assigned value.
- A variable store s ($Var \rightharpoonup \{true, false\}$), much like the While language.
- A big-step derivation rule of form $\langle B, s \rangle \Downarrow_b \langle s', b \rangle$ (program and store $\to$ final store and expression value).

We want determinacy and totality to be preserved, provide a suggestion of a rule that could be added to your solution to break either.

$$\text{(B-BOOL)}\frac{}{\langle b, s \rangle \Downarrow \langle s, b \rangle} \qquad \text{(B-NEG-FALSE)}\frac{\langle B, s \rangle \Downarrow_b \langle s', false \rangle}{\langle \neg B, s \rangle \Downarrow \langle s', true \rangle} \qquad \text{(B-NEG-TRUE)}\frac{\langle B, s \rangle \Downarrow_b \langle s', true \rangle}{\langle \neg B, s \rangle \Downarrow \langle s', false \rangle}$$

$$\text{(OR-SC)}\frac{\langle B_1, s \rangle \Downarrow_b \langle s', true \rangle}{\langle B_1 \vee B_2, s \rangle \vee \Downarrow \langle s', true \rangle} \qquad \text{(OR-EXH)}\frac{\langle B_1, s \rangle \Downarrow_b \langle s'', false \rangle \qquad \langle B_2, s'' \rangle \Downarrow_b \langle s', b \rangle}{\langle B_1 \vee B_2, s \rangle \Downarrow \langle s', b \rangle}$$

$$\text{(AND-SC)}\frac{\langle B_1, s \rangle \Downarrow_b \langle s', false \rangle}{\langle B_1 \wedge B_2, s \rangle \Downarrow \langle s', false \rangle} \qquad \text{(AND-EXH)}\frac{\langle B_1, s \rangle \Downarrow_b \langle s'', true \rangle \qquad \langle B_2, s'' \rangle \Downarrow_b \langle s', b \rangle}{\langle B_1 \wedge B_2, s \rangle \Downarrow \langle s', b \rangle}$$

$$\text{(ASSIGN)}\frac{\langle B, s \rangle \Downarrow_b \langle s'', b \rangle \qquad s' = [x \mapsto b]}{\langle x := B, s \rangle \Downarrow \langle s', b \rangle}$$

Hence we can now create derivations such as:

```
bool x;
(x = true) || (x = false);
```

$$\text{(OR-SC)}\frac{\text{(ASSIGN)}\frac{\text{(B-BOOL)}\frac{}{true \Downarrow true} \qquad (x \mapsto true) = ()[x \mapsto true]}{\langle x := true, () \rangle \Downarrow \langle (x \mapsto true), true \rangle}}{\langle (x := true) \vee (x := false), () \rangle \Downarrow \langle (x \mapsto true), true \rangle}$$

We can break determancy by adding short-circuiting rules for the right hand side (e.g $b \vee true \Downarrow true$) of $\vee$ and $\wedge$.

Q1a - 2021/22 **Exam Question 2.1.1**

Consider the language *GOTO*, comprising of the standard expressions E, boolean expressions B and the following commands (where $i, j \in \mathbb{N}$ are natural numbers):

$$C ::= exit \mid x := E \mid goto(i) \mid goto(B, i, j)$$

A *GOTO* program $P \in Prog$ is a map of numbers to commands:

$$P \in PROG \stackrel{def}{=} \mathbb{N} \to CMD \text{ where commands, } C \in CMD \text{ is defined as above}$$

Given a *GOTO* program P, $P(0)$ denotes the first command of P, $P(1)$ dnotes the second command of P, and so forth.

Using big-step operational semantics the expressions and booleans are evaluated against a (variable) store s as usual, and their evaluation is simplified so that the sore does not change.

$$\langle E, s\rangle \Downarrow_e n \text{ where } n \in \mathbb{N} \qquad \langle B, s\rangle \Downarrow_b b \text{ where } b \in \{true, false\}$$

Programs are also evaluated using a big-step operational semantics against a store s and the program counter $pc \in \mathbb{N}$ resulting in another store s' and a positive natural number $k \in \mathbb{N}^+$. That is the *GOTO* big-step operational semantics rules, given below are of form:

$$\langle P, s, pc\rangle \Downarrow \langle s', k\rangle$$

The rules are:

$$\text{(EXIT)}\frac{P(pc) = exit}{\langle P, s, pc\rangle \Downarrow \langle s, 1\rangle} \qquad \text{(JUMP)}\frac{P(pc) = goto(i) \qquad \langle P, s, i\rangle \Downarrow \langle s', k\rangle}{\langle P, s, pc\rangle \Downarrow \langle s', k+1\rangle}$$

$$\text{(ASSIGN)}\frac{P(pc) = x := E \qquad \langle E, s\rangle \Downarrow_e n \qquad \langle P, s[x \mapsto n], pc+1\rangle \Downarrow \langle s', k\rangle}{\langle P, s, pc\rangle \Downarrow \langle s', k+1\rangle}$$

$$\text{(BRANCH-TRUE)}\frac{P(pc) = goto(B, i, j) \qquad \langle B, s\rangle \Downarrow_e true \qquad \langle P, s, i\rangle \Downarrow \langle s' k\rangle}{\langle P, s, pc\rangle \Downarrow \langle s', k+1\rangle}$$

$$\text{(BRANCH-FALSE)}\frac{P(pc) = goto(B, i, j) \qquad \langle B, s\rangle \Downarrow_e false \qquad \langle P, s, j\rangle \Downarrow \langle s' k\rangle}{\langle P, s, pc\rangle \Downarrow \langle s', k+1\rangle}$$

Consider a program P with three instructions:

$$P(0) = x := x + 1$$
$$P(1) = goto(x > 0, 2, 0)$$
$$P(2) = exit$$

i) Give a derivation for $\langle P, s_0, 0\rangle \Downarrow \langle s_1, 3\rangle$ with $s_0 = [x \mapsto 0]$ and $s_1 = [x \mapsto 1]$.

   You may evaluate expressions and booleans directly without showing their derivation trees.

ii) Explain in words what k denotes when $\langle P, s, pc\rangle \Downarrow \langle s', k\rangle$.

iii) Explain in words the behaviour of the $goto(i)$ and $goto(B, i, j)$ commands.

iv) Define $goto(i)$ in terms of the other GOTO commands. You may use any GOTO command except $goto(i)$ in your definition.

Q1a - 2020/21 **Exam Question 2.1.2**

Consider the language *NONDET* comprising the standard expressions E, boolean expressions B, and the following commands.

$$C ::= skip \mid x := E \mid assume\ B \mid or(C, C) \mid loop(C) \mid C; C$$

Using a big-step operational semantics, the expressions and booleans are evaluated against a variable store s, and their evaluation is simplified so that the store does not change:

$$\langle E, s\rangle \Downarrow_e \ \text{ where } n \in \mathbb{N} \qquad \langle B, s\rangle \Downarrow_b b \text{ where } b \in \{true, false\}$$

Commands are also evaluated using a big-step operational semantics, against a variable store s, resulting in a new store s'. The big-step operational semantics rules of *NONDET* are given below:

$$\text{(SKIP)}\frac{}{\langle skip, s\rangle \Downarrow s} \qquad \text{(ASSIGN)}\frac{\langle E, s\rangle \Downarrow_e n \qquad s[x \mapsto n] = s'}{\langle x := E, s\rangle \Downarrow s'} \qquad \text{(ASSUME)}\frac{\langle B, s\rangle \Downarrow_b true}{\langle assume\ B, s \Downarrow s}$$

$$\text{(OR-LEFT)}\frac{\langle C_1, s\rangle \Downarrow s'}{\langle or(C_1, C_2), s\rangle \Downarrow s'} \qquad \text{(OR-RIGHT)}\frac{\langle C_2, s\rangle \Downarrow s'}{\langle or(C_1, C_2), s\rangle \Downarrow s'}$$

$$\text{(LOOP-ITER)}\frac{\langle C, s\rangle \Downarrow s'' \qquad \langle loop(C), s''\rangle \Downarrow s'}{\langle loop(C), s\rangle \Downarrow s'} \qquad \text{(LOOP-EXIT)}\frac{}{\langle loop(C), s\rangle \Downarrow s}$$

$$\text{(SEQ)}\frac{\langle C_1, s\rangle \Downarrow s'' \qquad \langle C_2, s''\rangle \Downarrow s'}{\langle C_1; C_2, s\rangle \Downarrow s'}$$

i) Give the derivation tree corresponding to the the big-step derivation $\langle C, s_0\rangle \Downarrow s_2$ where:

$$C = loop(x := x + 1) \qquad s_0 = [x \mapsto 0] \qquad s_2 = [x \mapsto 2]$$

You may evaluate expressions and booleans directly, without showing their derivation trees.

ii) Explain in words the behaviour of the *loop* command.

iii) Let $\langle C, s_0\rangle \Downarrow s'$ for some store s' where C and s_0 are defined as in part i.

What are the possible values of x in s'? Justify your answer in words.

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

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

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

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 2.1.4**

Add a rule to break determinacy without breaking confluence.

$$(\text{S-RIGHT-E})\frac{E_2 \to E_2'}{E_1 + E_2 \to E_1 + E_2'}$$

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

Q1b - 2020/21 **Exam Question 2.1.3**

...continued from Q1a - 2020/21

Give the small-step operational semantics rules for $or(C_1, C_2)$ and $loop(C)$.

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:

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

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

(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

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

Q1b - 2021/22 | **Exam Question 2.2.1**

... continued from Q1a - 2021/22

We can similarly define the small-step operational semantics of *GOTO* programs to be of the form $P, s, pc \to P, s', pc'$ where s and pc are the starting store and program counter, and s' and pc' are the resulting store and program counter. For instance, for $x := E$ we have:

$$\text{(ASSIGN)}\frac{P(pc) = x := E \qquad \langle E, s\rangle \Downarrow_e n \qquad s' = [x \mapsto n] \qquad pc' = pc + 1}{P, s, pc \to P, s', pc'}$$

Note that for simplicity we use the big-step evaluation of expressions in the premise above. You may use big step evaluation rules for expressions (including booleans) in your answer.

Give the small-step operational semantics rules for $goto(i)$ and $goto(B, i, j)$.

2.2.4 Properties

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

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

Hence the relation is also confluent:

$$\forall C, C_1, C_2 \in Com \forall s, s_1, s_2.[\langle C, s\rangle \to_c \langle C_1, s_1\rangle \wedge \langle C, s\rangle \to_c \langle C_2, s_2\rangle \to$$

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

Both also hold for $\to_e$ and $\to_b$.

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.

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

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 $\to_b$ and $\to_e$ are normalising, but $\to_c$ is not as it may not have a normal form.

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

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

($\to_c^3$ 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:

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

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

(value depends on evaluation order)

2.2.8 Short Circuit Semantics

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

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.

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

We can evaluate $Factorial$ 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 $Factorial$ 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 $s = (x \mapsto 3, y \mapsto 17, z \mapsto 42)$:

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

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 $s = (x \mapsto 3, y \mapsto 17, z \mapsto 42)$:

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

Result:

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

Eliminate skip

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

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 1)$:

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

Result:

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

Eliminate Skip

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

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

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

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 $s = (x \mapsto 3, y \mapsto 3, z \mapsto 42, a \mapsto 3)$:

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

Result:

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

Eliminate skip

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

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

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 $plus(zero, zero) = zero$

$$\begin{array}{llll} (1) & \text{LHS} & = & plus(zero, zero) & \\ (2) & & = & zero & \text{(By definition of } plus\text{)} \\ (3) & & = & \text{RHS} & \text{(As Required)} \end{array}$$

Inductive Case

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

$$\begin{array}{lllll} (1) & \text{LHS} & = & plus(succ(K), zero) & \\ (2) & & = & succ(plus(K, zero)) & \text{(By definition of } plus\text{)} \\ (3) & & = & succ(K) & \text{(By Inductive Hypothesis)} \\ (4) & & = & \text{RHS} & \text{(As Required)} \end{array}$$

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*:

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

I speak for the trees. . . **Example Question 3.1.1**

Prove 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:

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

Lemma ⇒ Confluence

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

(1)	Take some arbitrary $E, E_1, E_2 \in SimpleExp$, 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:

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

There are 3 rules that apply:

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

Base Case

Take arbitrary $n \in \mathbb{N}$ and $E_1, E_2 \in SimpleExp$ such that $n \rightarrow E_1 \land n \rightarrow E_2$ 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)	$\neg(n \rightarrow E_1 \land n \rightarrow E_2)$	(By 2)
(4)	$n \rightarrow E_1 \land n \rightarrow E_2 \Rightarrow E_1 = E_2$	(By 3)
(5)	$E \rightarrow E_1 \land E \rightarrow E_2 \Rightarrow E_1 = E_2$	(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 $E_3, E_4 \in SimpleExp$ 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 $E' \in SimpleExp$ such that $E_2 \rightarrow E'$	(By case B)
(3)	$E_3 = n + E'$	(By case B)
(4)	There exists $E'' \in SimpleExp$ 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)

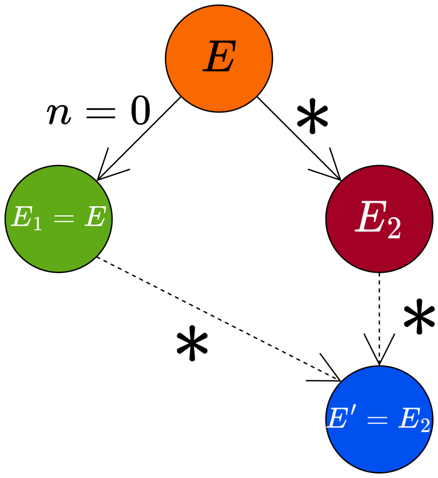
For C:

(1)	There exists $E' \in \mathit{SimpleExp}$ such that $E_1 \rightarrow E'$	(By case C)
(2)	There exists $E'' \in \mathit{SimpleExp}$ such that $E_1 \rightarrow E''$	(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.)

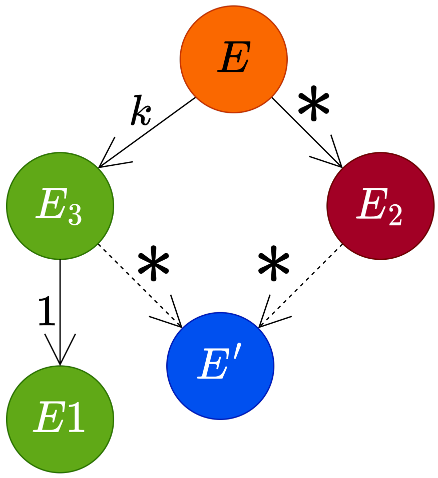
Base Case

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

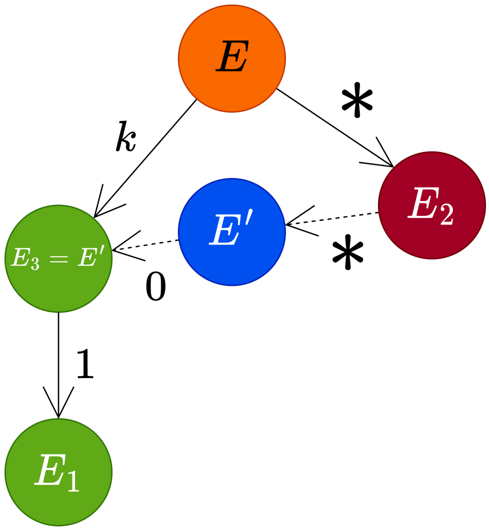


Inductive Case

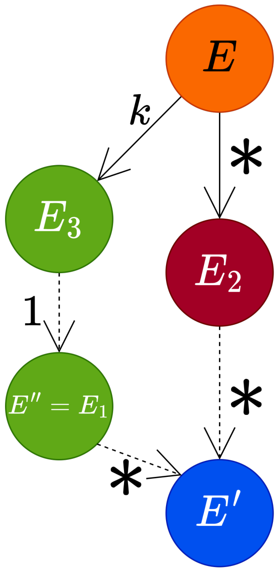
Next we assume confluence for up to k steps, and attempt to prove for $k + 1$ steps.



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



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



Q1 c - 2021/22 **Exam Question 3.2.1**

. . . continued from Q1b - 2021/22

Let us assume the following evaluation rules are deterministic.

$$\forall b_1, b_2, B, s.\ [\langle B, s\rangle \Downarrow_b b_1 \wedge \langle B, s\rangle \Downarrow_b b_2 \Rightarrow b_1 = b_2] \qquad \text{(BOOl-DET)}$$

$$\forall n_1, n_2, E, s.\ [\langle E, s\rangle \Downarrow_e n_1 \wedge \langle E, s\rangle \Downarrow_e n_2 \Rightarrow n_1 = n_2] \qquad \text{(EXPR-DET)}$$

Prove that the program evaluation rules are deterministic.

$$\forall k_1, k_2, s_1, s_2, P, s, pc.\ [\langle P, s, pc\rangle \Downarrow \langle s_1, k_1\rangle \wedge \langle P, s, pc\rangle \Downarrow \langle s_2, k_2\rangle \Rightarrow s_1 = s_2 \wedge k_1 = k_2]$$

Do you proof using mathematical induction on k_1. You may use assumptions (BOOL-DET) and (EXPR-DET) in your proof.

Q1c - 2020/21 **Exam Question 3.2.2**

. . . continued from Q1a - 2020/21

Recall the *WHILE* language from lectures. We can annotate the big-step operational semantics of *WHILE* to record the derivation depth $i \in \mathbb{N}$. This is just a simple annotation that will help with the proofs. Formally:

$$\langle C, s\rangle \Downarrow_i s' \text{ where } i \in \mathbb{N}$$

The annotated big-step operational semantics of $WHILE$ are given below.

$$(\text{SKIP})\frac{}{\langle skip, s\rangle \Downarrow_0 s} \qquad (\text{ASSIGN})\frac{\langle E, s\rangle \Downarrow_e n \qquad s[x \mapsto n] = s'}{\langle x := E, s\rangle \Downarrow_0 s'}$$

$$(\text{IF-TRUE})\frac{\langle B, s\rangle \Downarrow_b true \qquad \langle C_1, s\rangle \Downarrow_i s'}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \Downarrow_{i+1} s'} \qquad (\text{IF-TRUE})\frac{\langle B, s\rangle \Downarrow_b false \qquad \langle C_2, s\rangle \Downarrow_i s'}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \Downarrow_{i+1} s'}$$

$$(\text{WHILE-ITER})\frac{\langle B, s\rangle \Downarrow_b true \qquad \langle C, s\rangle \Downarrow_i s'' \qquad \langle \text{while } B \text{ do } C, s''\rangle \Downarrow_j s' \qquad k = max(i,j)}{\langle \text{while } B \text{ do } C, s\rangle \Downarrow_{k+1} s'}$$

$$(\text{WHILE-BREAK})\frac{\langle B, s\rangle \Downarrow_b false}{\langle \text{while } B \text{ do } C, s\rangle \Downarrow_0 s} \qquad (\text{SEQ})\frac{\langle C_1, s\rangle \Downarrow_i s'' \qquad \langle C_2, s''\rangle \Downarrow_j s' \qquad k = max(i,j)}{\langle C_1; C_2, s\rangle \Downarrow_{k+1} s'}$$

Consider the translation function f from $WHILE$ commands to $NONDET$ commands, defined inductively as follows:

$$\begin{aligned} f(skip) &= skip \\ f(x := E) &= x := E \\ f(\text{if } B \text{ then } C_1 \text{ else } C_2) &= or((assume\ B; f(C_1)), (assume\ \neg B; f(C_2))) \\ f(\text{while } B \text{ do } C) &= loop(assume\ B; f(C)); assume \neg B \\ f(C_1; C_2) &= f(C_1); f(C_2) \end{aligned}$$

Prove that the translation f preserves the meaning of commands:

$$\forall i, C, s, s'. \ [\langle C, s\rangle \Downarrow_i s' \Rightarrow (f(C), s) \Downarrow s']$$

Do your proof using strong mathematical induction on i. You may also use the following lemma:

$$\forall B, s. \ [\langle B, s\rangle \Downarrow_b false \Rightarrow \langle \neg B, s\rangle \Downarrow_b true] \qquad (\text{LEMMA-EXCLUDED-MIDDLE})$$

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

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 $E_1 \to^0 E_1'$.

(1)	$E_1 = E_1'$	(By definition of $\to^0$)
(2)	$(E_1 + E_2) = (E_1' + E_2)$	(By 1)
(3)	$(E_1 + E_2) \to^0 (E_1' + E_2)$	(By definition of $\to^0$)

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

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

3.3.5 Determinacy of $\to$ for Exp

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

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

Determinacy:

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

We prove this using property P:

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

Base Case: $E = x$

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

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

Base Case: $E = x$

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

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

. . . Inductive Step . . .

3.3.6 Syntax of Commands

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

Determinacy

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

Confluence

$$\forall C, C_1, C_2 \in \mathit{Com}.\forall s, s_1, s_2 \in \mathit{State}.[\langle C, s\rangle \to_c^* \langle C_1, s_1\rangle \wedge \langle C, s\rangle \to_c^* \langle C_2, s_2\rangle \Rightarrow \exists C' \in \mathit{Com}.\exists s' \in \mathit{State}.$$
$$[\langle C_1, s_1\rangle \to_c^* \langle C', s'\rangle \wedge \langle C_2, s_2\rangle \to_c^* \langle C', s'\rangle]]$$

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

Register Machine Simulator ***Extra Fun!* 4.0.1**

Register Machine Simulator   Repository

This simulator has been developed by Yìtáng Chén to support 50003, make sure to give him a ★!

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.

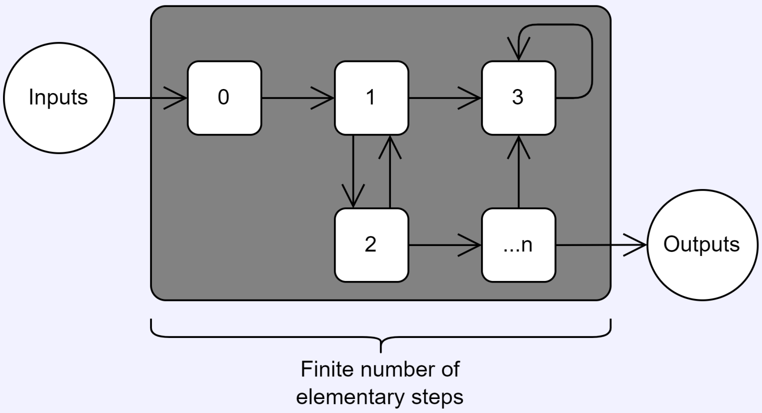
It was proven that no such algorithm exists by Alonzo Church and Alan Turing using their notions of Computing which show it is not computable.

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.

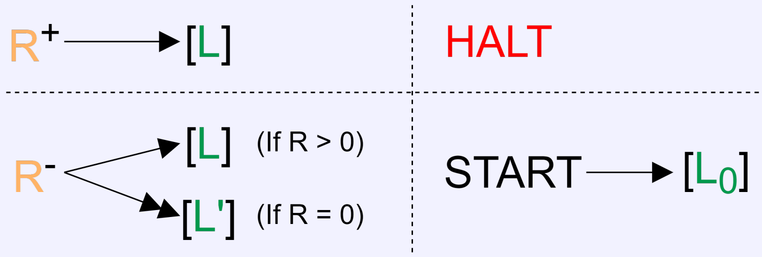
- Infinitely many registers, each storing a natural number ($\mathbb{N} \triangleq \{0, 1, 2, \dots\}$)
- Each instruction has a label associated with it.

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:

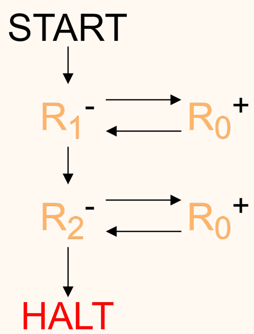
$$f(x, y, z) = x + y + z$$

Registers

$R_0 \quad R_1 \quad R_2$

Program

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



Example 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.

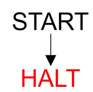
4.2.2 Computable Functions

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

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

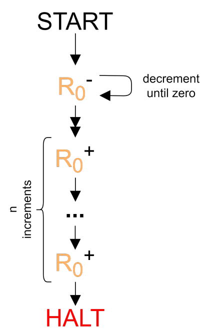
HALT



Constant

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

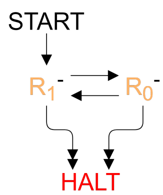
$$\begin{array}{ll} L_0: & R_0^- \to L_0, L_1 \\ L_1: & R_0^+ \to L_2 \\ \vdots & \vdots \\ L_n: & R_0^+ \to L_{n+1} \\ L_{n+1}: & \textbf{HALT} \end{array}$$



Truncated Subtraction

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

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

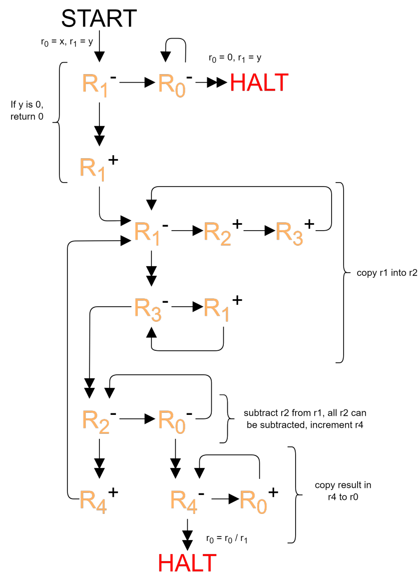


Integer Division

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).

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

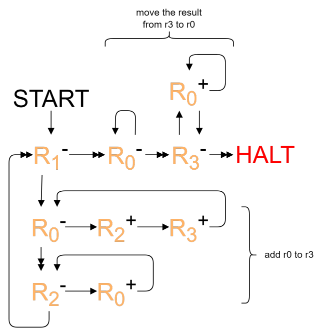
$$\begin{array}{ll} L_0: & R_1^- \to L_3, L_2 \\ L_1: & R_0^- \to L_1, L_2 \\ L_2: & \textbf{HALT} \\ L_3: & R_1^+ \to L_4 \\ L_4: & R_1^- \to L_5, L_7 \\ L_5: & R_2^+ \to L_6 \\ L_6: & R_3^+ \to L_4 \\ L_7: & R_3^- \to L_8, L_9 \\ L_8: & R_1^+ \to L_9 \\ L_9: & R_2^- \to L_{10}, L_4 \\ L_{10}: & R_0^-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



Multiplication

$$x \times y$$

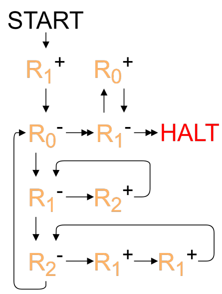
$$\begin{array}{ll} L_0: & R_1^- \to L_5, L_1 \\ L_1: & R_0^- \to L_1, L_2 \\ L_2: & R_3^- \to L_3, L_4 \\ L_3: & R_0^+ \to L_2 \\ L_4: & \textbf{HALT} \\ L_5: & R_0^- \to L_6, L_8 \\ L_6: & R_2^+ \to L_7 \\ L_7: & R_3^+ \to L_5 \\ L_8: & R_2^- \to L_9, L_0 \\ L_9: & R_0^+ \to L_8 \end{array}$$



Exponent of base 2

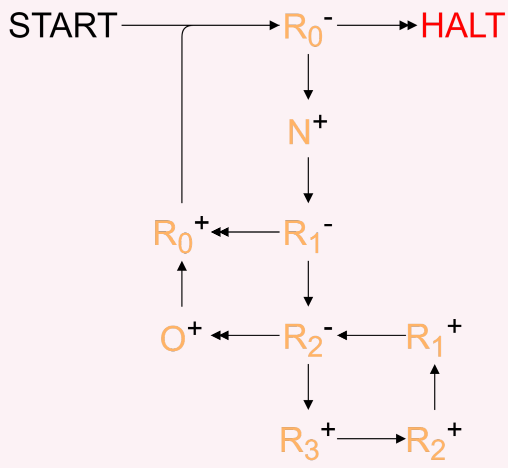
$$e(x) \triangleq 2^x$$

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



Q2bi - 2021/22 **Exam Question 4.2.1**

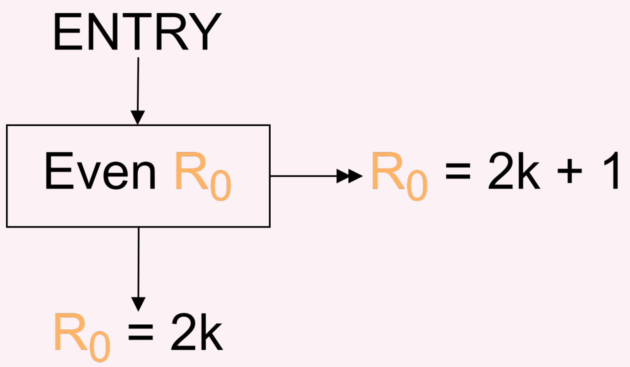
Consider the graphical representation of a Register Machine M:



Write down the program/code (list of instructions) for this machine using only a single **HALT** instruction (at the end of the code).

Q2bi - 2020/21 **Exam Question 4.2.2**

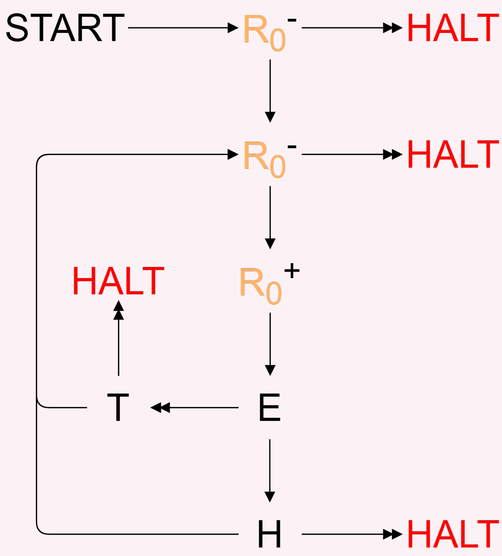
Describe (graphically) a Register Machine (RM) gadget which tests if R_0 is even or odd without changing the value of R_0 and using only RM instructions (no gadgets).



Q2bii - 2020/21 **Exam Question 4.2.3**

Note: This question contains corrections from the original paper

Consider the graphical representation of register machine M:



Take for $E \equiv R_1^-$, $H \equiv R_2^-$ and $T \equiv R_0^-$.

Write down the program, code or list of instructions for this machine using only a single **HALT** instruction (at end of the code).

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

$$\begin{array}{llll} \langle\langle x,y\rangle\rangle & = 2^x(2y+1) & y\ 1\ 0_1\dots 0_x & \text{Bijection between } \mathbb{N}\times\mathbb{N} \text{ and } \mathbb{N}^+ = \{n\in\mathbb{N} \mid n\neq 0\} \\ \langle x,y\rangle & = 2^x(2y+1)-1 & y\ 0\ 1_1\dots 1_x & \text{Bijection between } \mathbb{N}\times\mathbb{N} \text{ and } \mathbb{N} \end{array}$$

Q2a - 2021/22 **Exam Question 4.3.1**

Either state your birthday or take today's date as $B = YYMMDD$ (i.e. last two digits of the year, two digits representing month and day each) and determine the pair, the list, and the Register Machine (RM) instruction it represents, i.e. for which pair x, y do we have $\langle\langle x, y\rangle\rangle = B$, for which list ℓ of numbers do we get $\ulcorner \ell \urcorner = B$, and for which Register Machine instruction I do we have that $\ulcorner I \urcorner = B$?

Show your work, e.g. binary representation of your B, etc.

Q2a - 2020/21 **Exam Question 4.3.2**

State your CID and determine the pair, the list, and the Register Machine (RM) instruction it represents, i.e. for which pair x, y do we have $\langle\langle x, y\rangle\rangle = \text{CID}$, for which list ℓ of numbers do we get $\ulcorner ell \urcorner = CID$, and for which register-machine instructions I do we have that $\ulcorner I \urcorner = CID$?

Show your work, e.g. binary representation of your CID, etc.

Add eight to your CID, i.e. consider $CID + 8$, and repeat these three decodings.

Can one be sure that very student in class can (in principle) decode their CIDs as requested?

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

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

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

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

4.3.4 Programs

Given some program:

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

Tools ***Extra Fun!* 4.3.1**

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.

Have a go at making your own register machine encode/decode and simulation in your language of choice!

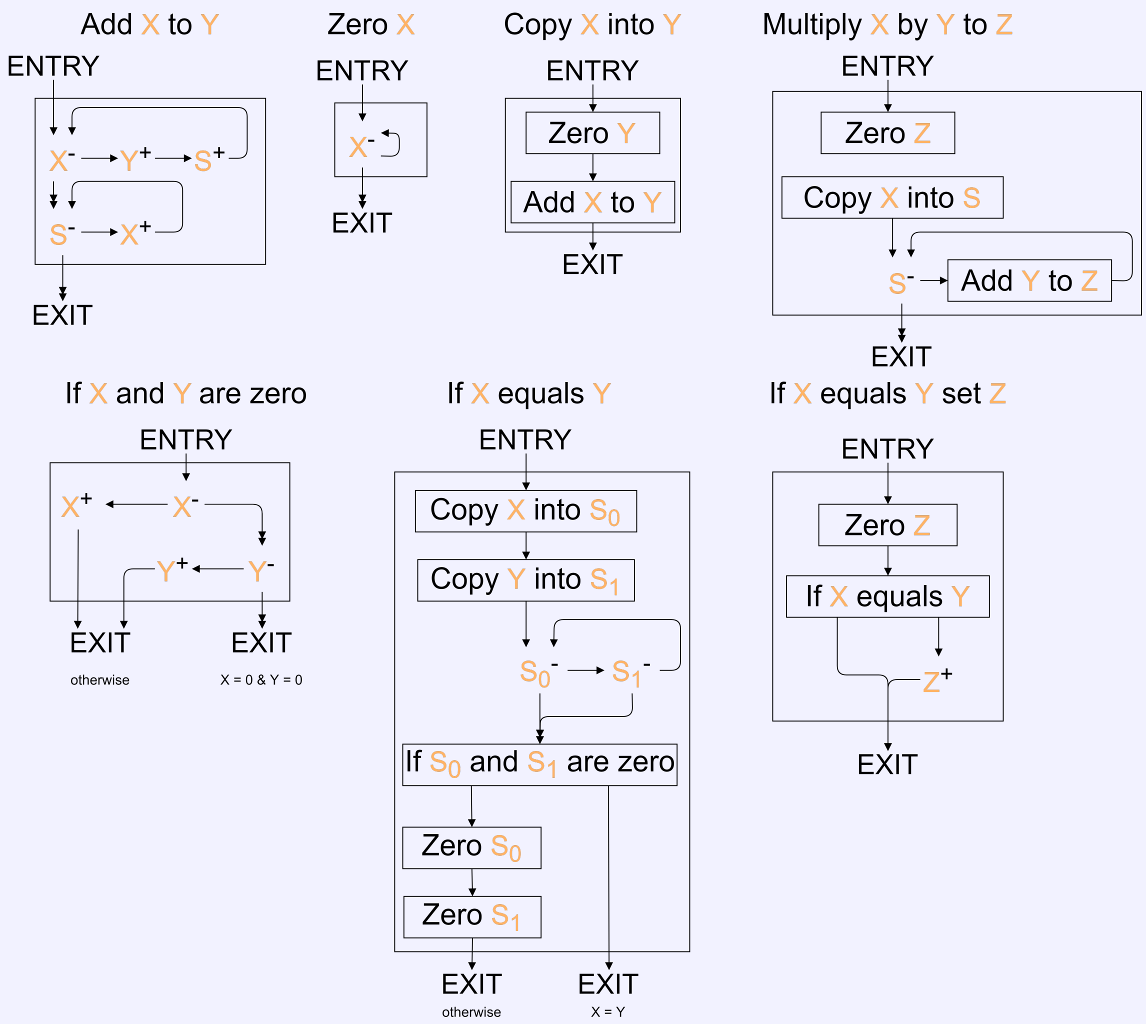
4.4 Gadgets

Register Machine Gadget **Definition 4.4.1**

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* → *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.

Q2bii - 2021/22 **Exam Question 4.4.1**

. . . continued from question Q2bi - 2021/22

Replace some instructions by gadgets as follows:

R_0^+ by copy N to O	R_1^- by pop O to R_1	R_2^+ by push R_1 to N
N^+ by copy Y to N	R_2^- by pop O to R_2	R_1^+ by copy R_2 to R_1
O^+ by push X to O	R_3^+ by add R_2 to R_1	

where for pop gadgets the *empty* exit is identified with $\twoheadrightarrow$ and *done* with $\rightarrow$ and where we have additional registers: X with (constant) value 1 and Y with (constant) value 2.

Draw the graphical representation of the resulting RM and describe its execution with initially: $R_0 = 3$ and all other registers set to 0 (except for X and Y), use the same labels as in the original RM.

Q2biii - 2021/22 **Exam Question 4.4.2**

...continued from Q2bii - 2021/22

What does this register machine compute for $R_0 = n$ and all other registers set to 0 (except for X and Y), i.e. what does the contents of register N or O represent when the RM terminates?

Give an interpretation of what the registers are used for/hold.

4.5 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:

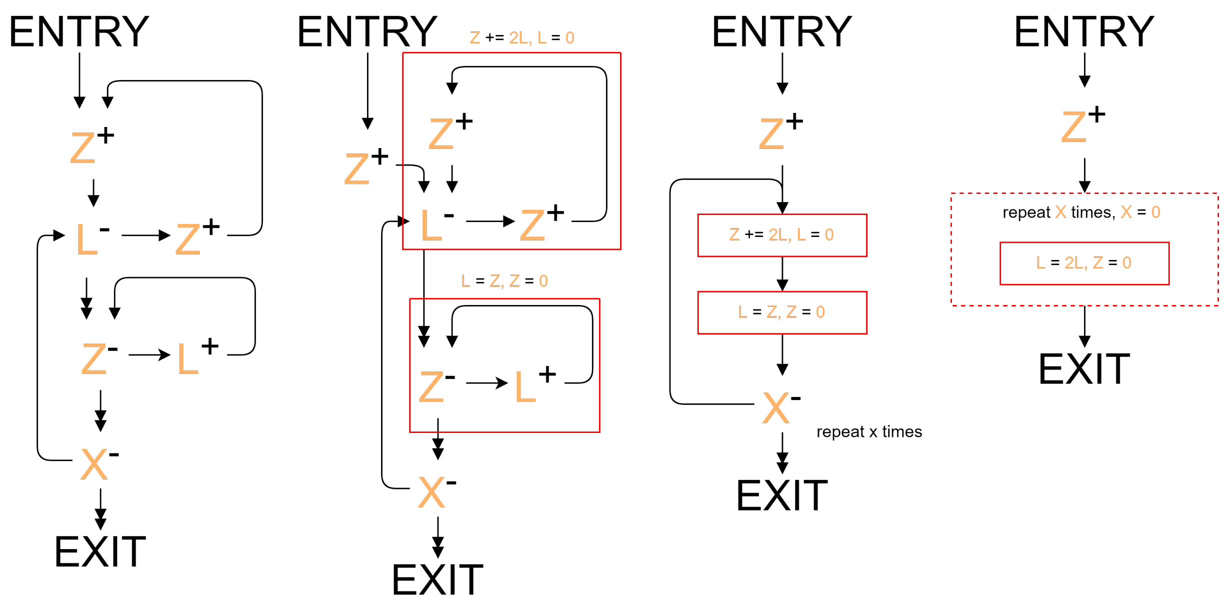
4.5.1 Experimentation

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

4.5.2 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 $L = 2^X(2L + 1)$

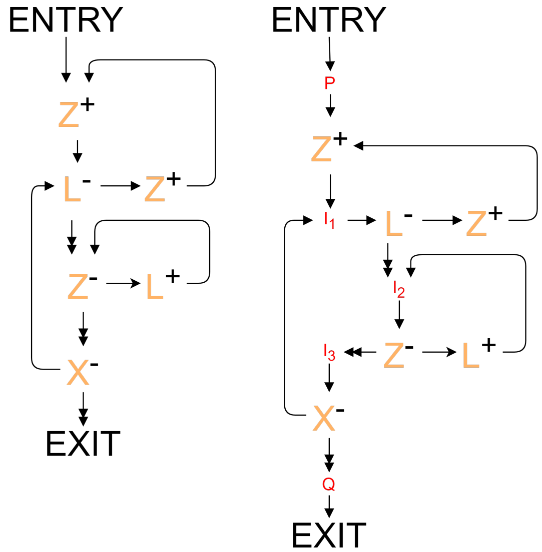


4.5.3 Invariants

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:

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

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.

4.6 Universal Register Machine

A register machine that simulates a register machine.

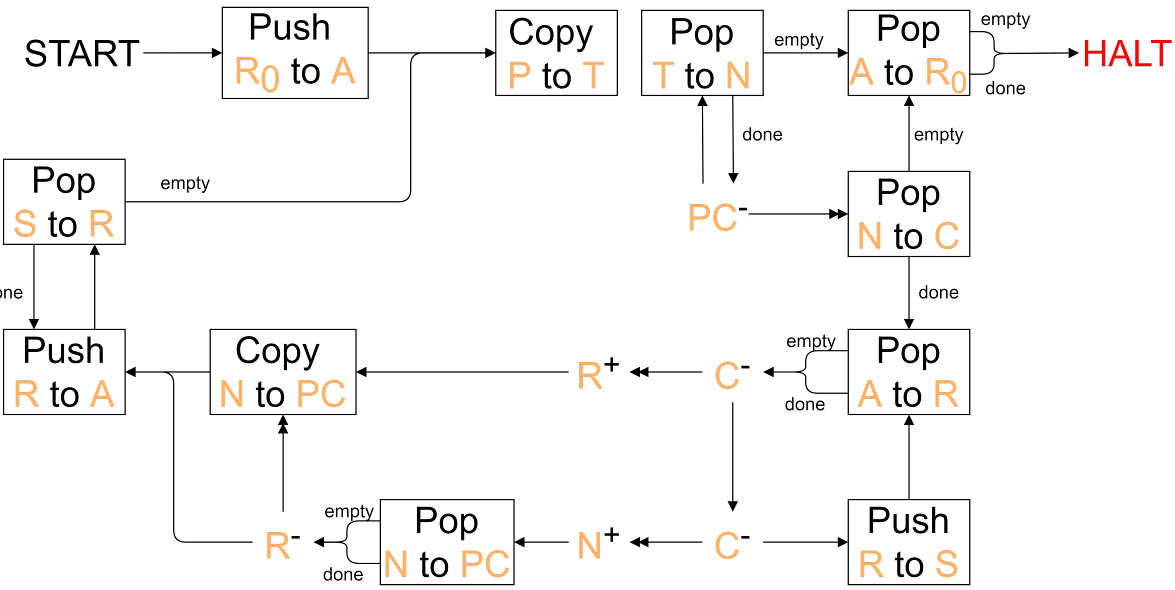
It takes the arguments:

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
$R_9 \dots$		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
```


START
Push
R0 to A
Copy
P to T
Pop
T to N
empty
Pop
A to R0
empty
done
HALT
done
empty
Pop
S to R
empty
PC-
Pop
N to C
done
done
Push
R to A
Copy
N to PC
R+
C-
empty
done
Pop
A to R
empty
Pop
N to PC
R-
done
N+
C-
Push
R to S

Chapter 5

Halting Problem

5.1 Halting Problem for Register Machines

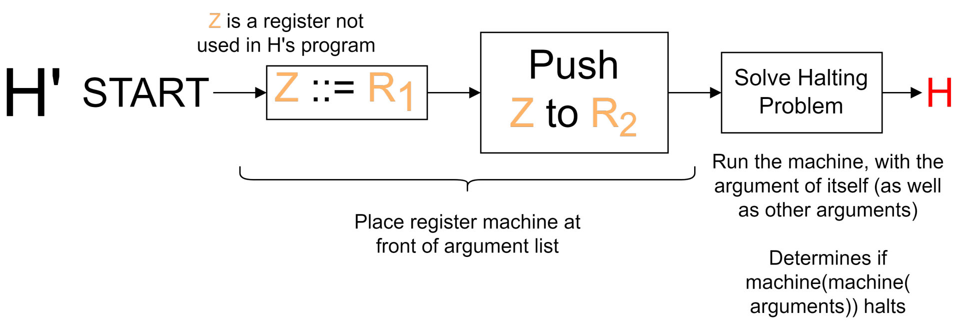
A register machine H decides the halting problem if for all $e, a_1, \ldots, a_n \in \mathbb{N}$:

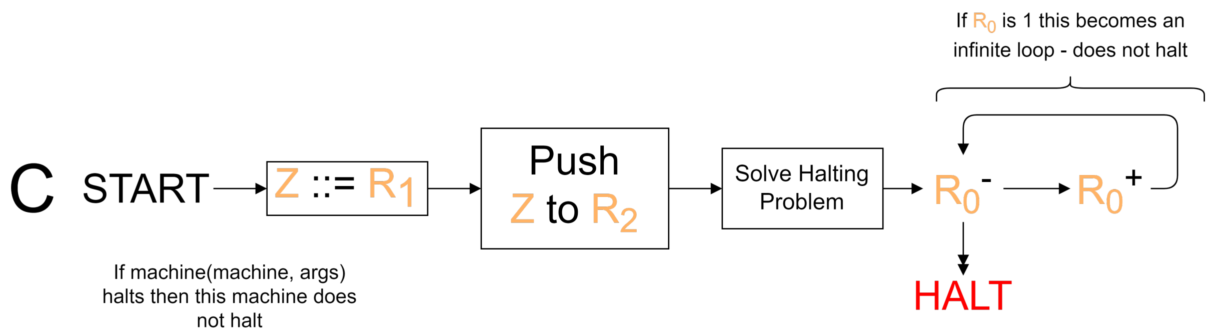
$$R_0 = 0 \quad R_1 = e \quad R_2 = \ulcorner [a_1, \ldots, a_n] \urcorner \quad R_{3..} = 0$$

And where H halt with the state as follows:

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

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





Hence when we run C with the argument C we get a contradiction.

$C(C)$ **Halts**	Then C with $R_1 = \ulcorner C \urcorner$ as an argument does not halt, which is a contradiction
$C(C)$ **Does not Halt**	Then C with $R_1 = \ulcorner C \urcorner$ as an argument halts, which is a contradiction

5.2 Computable Functions

5.2.1 Enumerating the Computable Functions

Onto (Surjective) **Definition 5.2.1**

Each element in the codomain is mapped to by at least one element in the domain.

$$\forall y \in Y. \exists x \in X. [f(x) = y] \Rightarrow f \text{ is onto}$$

For each $e \in \mathbb{N}$, $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ (partial function computed by $program(e)$):

$$\varphi_e(x) = y \Leftrightarrow program(e) \text{ with } R_0 = 0 \land R_1 = x \text{ halts with } R_0 = y$$

Hence for a given program $\in \mathbb{N}$ we can get the computable partial function of the program.

$$e \mapsto \varphi_e$$

Therefore the above mapping represents an *onto/surjective* function from $\mathbb{N}$ to all computable partial functions from $\mathbb{N} \rightharpoonup \mathbb{N}$.

5.2.2 Uncomputable Functions

For $f : X \rightharpoonup Y$ (partial function from X to Y):

$$f(x)\uparrow \triangleq \neg\exists y \in Y. [f(x) = y]$$
$$f(x)\downarrow \triangleq \exists y \in Y. [f(x) = y]$$

Hence we can attempt to define a function to determine if a function halts.

$$f \in \mathbb{N} \rightharpoonup \mathbb{N} \triangleq \{(x, 0) | \varphi_x(x)\uparrow\} \triangleq f(x) = \begin{cases} 0 & \varphi_x(x)\uparrow \\ undefined & \varphi_x(x)\downarrow \end{cases}$$

However we run into the halting problem:

Assume f is computable, then $f = \varphi_e$ for some $e \in \mathbb{N}$.

if $\varphi_e(e)\uparrow$ by definition of f, $\varphi_e(e) = 0$ so $\varphi_e(e)\downarrow$ which is a contradiction
if $\varphi_e(e)\downarrow$ by definition of f, $f(e)\uparrow$, and hence as $f = \varphi_e$, $\varphi_e\uparrow$ which is a contradiction

Here we have ended up with the halting problem being uncomputable.

Collatz Conjecture ***Extra Fun!* 5.2.1**

A famous example of a simple algorithm not yet determined to terminate on all inputs.

Given some input n, how many steps of applying f are required to reach 1*, given:*

$$f = \begin{cases} \frac{n}{2} & n \text{ is even} \\ 3n + 1 & n \text{ is odd} \end{cases}$$

The conjecture states that the sequence from any positive integer n will eventually go to zero. And hence any algorithm generating the sequence will terminate. This remains unproven.

```
-- get the sequence for positive integer
-- note we have use Integral (as proven to terminate for all values of fixed size Int)
collatz :: (Integral a) => a -> [a]
collatz 1 = [1]
collatz n
    | odd n     = n : collatz (n * 3 + 1)
    | otherwise = n : collatz (n `div` 2)

limit :: (Integral a) => a -> Int
limit = length . collatz
```

5.2.3 Undecidable Set of Numbers

Given a set $S \subseteq \mathbb{N}$, its characteristic function is:

$$\chi_S \in \mathbb{N} \to \mathbb{N} \quad \chi_S(x) \triangleq \begin{cases} 1 & x \in S \\ 0 & x \notin S \end{cases}$$

S is *register machine decidable* if its characteristic function is a register machine computable function.

S is decidable iff there is a register machine M such that for all $x \in \mathbb{N}$ when run with $R_0 = 0, R_1 = x$ and $R_{2..} = 0$ it eventually halts with:

$$R_0 = 1 \Leftrightarrow x \in S \qquad R_0 = 1 \Leftrightarrow x \notin S$$

Hence we are effectively asking if a register machine exists that can determine if any number is in some set S.

We can then define subsets of $\mathbb{N}$ that are decidable/undecidable.

The set of functions mapping 0 is undecidable

Given a set:

$$S_0 \triangleq \{e \mid \varphi_e(0) \downarrow\}$$

Hence we are finding the set of the indexes (numbers representing register machines) that halt on input 0.

If such a machine exists, we can use it to create a register machine to solve the halting problem. Hence this is a contradiction, so the set is undecidable.

The set of total functions is undecidable

Take set $S_1 \subseteq \mathbb{N}$:

$$S_1 \triangleq \{e \mid \varphi_e \text{total function}\}$$

If such a register machine exists to compute χ_{S_1}, we can create another register machine, simply checking 0. Hence as from the previous example, there is no register machine to determine S_0, there is none to determine S_1.

Q2biii - 2020/21 **Exam Question 5.2.1**

... continued from Q2bii - 2020/21

Take the machine M in (ii) and use the substitutions to create register machine K:

$E \triangleq even\ R_0$	exiting by $\twoheadrightarrow$ if odd, or by $\to$ if even.
$H \triangleq R_0 := R_0/2$	exiting by $\twoheadrightarrow$ if odd, or by $\to$ if even.
$T \triangleq R_0 := 3 \times R_0 + 1$	with no $\twoheadrightarrow$ needed.

Describe what the Register Machine K computing? In particular sketch an execution of K with initially $R_0 = 0, 1, 2, 3, 4$ and 5.

Q2biv - 2020/21 **Exam Question 5.2.2**

... continued from Q2biii - 2020/21

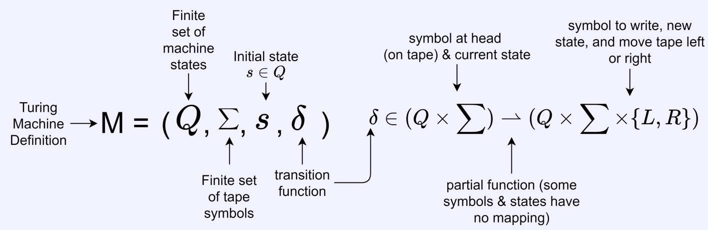
To the best of our knowledge nobody could yet show that K halts for all possible initial values of R_0. How does this relate to the Halting problem for RMs?

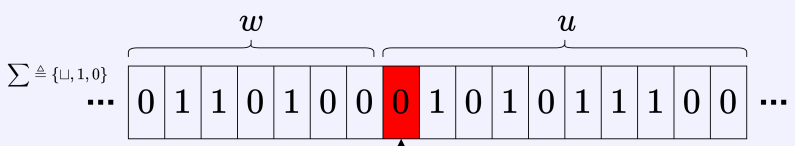
Chapter 6

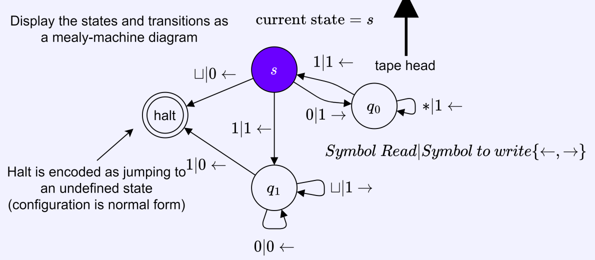
Turing Machines

6.1 Definition

Turing Machine **Definition 6.1.1**





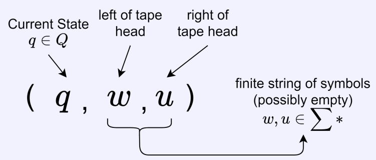


We can display the transition function as a table

	$\sqcup$	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 mean undefined, which results in a halt

Configuration:



Start State: With the start state, no symbols to the left and the input symbols to the right

(s, ϵ, u) where $\epsilon =$ empty

Normal form: when there is no transition for the current state and symbol at the tape head

(q, w, u) where $\delta(q, first\ u) \uparrow$

Transition:

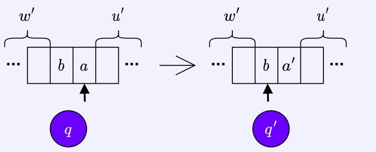
Note: When matching on an empty list, we get a new, empty element

$\sqcup : [\,] = [\,]$

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

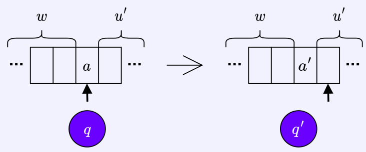
Transition moves to the left

$$\frac{a : u' = u,\ b : w' = w,\ \delta(q, a) = (q', a', L)}{(q, w, u) \to_M (q', w', b : a' : u')}$$



Transition moves to the right

$$\frac{a : u' = u,\ \delta(q, a) = (q', a', R)}{(q, w, u) \to_M (q', w : a', u')}$$



$$c_0 \to_M c_1 \to_M c_2 \to_M \cdots \to_M \begin{cases} c_n & \text{finite sequence - the turing machine halts} \\ \dots & \text{infinite sequence - turing machine does not halt} \end{cases}$$

Register machines abstract away the representation of numbers and operations on numbers (just uses $\mathbb{N}$ with increment, decrement operations), *Turing machines* are a more concrete representation of computing.

6.1.1 Turing $\to$ Register Machine

We can show that any computation by a *Turing Machine* can be implemented by a *Register Machine.* Given a *Turing Machine* M:

1. Create a numerical encoding of M's finite number of states, tape symbols, and initial tape contents.
2. Implement the transition table as a register machine.
3. Implement a register machine program to repeatedly carry out $\to_M$

Hence *Turing Machine Computable* $\Rightarrow$ *Register Machine Computable.*

Turing Machine Number Lists

In order to take arguments, and return value we need to encode lists on number on the tape of a turing machine. This is done as strings of unary values.

$$= \{\sqcup, 0, 1\} \quad \overbrace{\underbrace{\dots}_{\text{all } \sqcup} \;\; 0 \;\; \underbrace{1\dots1}_{n_1} \;\; \sqcup \;\; \underbrace{1\dots1}_{n_2} \;\; \sqcup \;\; \dots \;\; \sqcup \;\; \underbrace{1\dots1}_{n_k} \;\; 0 \;\; \underbrace{\dots}_{\text{all } \sqcup}}^{\text{Turing Machine Tape}}$$

Q2c - 2021/22 **Exam Question 6.1.1**

Specify a turing machine $M = (\mathcal{Q}, \Sigma, s, \sigma)$ which takes a tape with the representation of a list $\ell = [x_1, x_2, \dots, x_n]$ and terminates with a tape representing the singleton list $[s]$ where:

$$s = \left(\sum_{i=1}^{n} x_i\right) + n$$

Describe the computational steps (configurations) of M when the initial tape represents the list $[1, 2, 3]$.

Turing Computable **Definition 6.1.2**

If $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *Turing Computable* iff there is a turing machine M such that:

From initial state $(s, \epsilon, [x_1, \dots, x_n])$ (tape head at the leftmost 0), M halts if and only if $f(x_1, \dots, x_n)\downarrow$, and halts with the tape containing a list, the first element of which is y such that $f(x_1, \dots, x_n) = y$.

More formally, given $M = (Q, \sum, s, \delta)$ to compute f:

$$f(x_1, \dots, x_n)\downarrow \wedge f(x_1, \dots, x_n) = y \Leftrightarrow (s, \epsilon, [x_1, \dots, x_n]) \to_M^* (*, \epsilon, [y, \dots])$$

Register $\to$ Turing Machine

It is also possible to simulate any register machine on a turing machine. As we can encode lists of numbers on the tape, we can simply implement the register machine operations as operations on integers on the tape.

Hence *Register Machine Computable* $\Rightarrow$ *Turing Machine Computable.*

Notions of Computability

Every computable algorithm can be expressed as a turing machine (*Church-Turing Thesis*). In fact *Turing Machines*, *Register Machines* and the *Lambda Calculus* are all equivalent (all determine what is computable).

- **Partial Recursive Functions** Godel and Kleene (1936)
- **λ-Calculus** Church (1936)

- **canonical systems for generating the theorems of a formal system** Post (1943) and Markov (1951)
- **Register Machines** Lambek and Minsky (1961)
- **And many more . . .** (multi-tape turing machines, parallel computation, turing machines embeded in cellular automata etc)

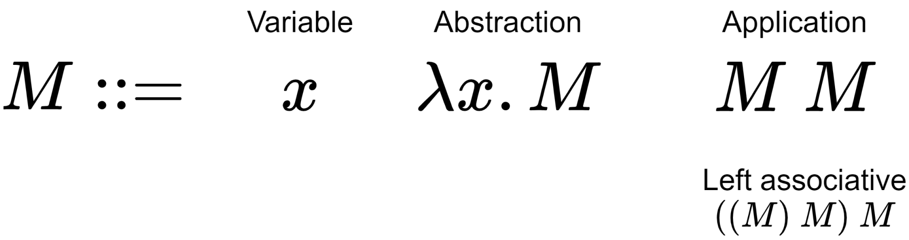
Chapter 7

Lambda Calculus

Type Systems for Programming Languages — ***Extra Fun!* 7.0.1**

The third-year type systems module contains an introduction to lambda calculus that can be found here.

7.1 Lambda Calculus



7.2 Syntax

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

7.2.1 Bound and Free Formally

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

α-equivalence — **Definition 7.2.1**

$$M =_\alpha N \text{ if and only if } N \text{ can be obtained from } M \text{ by renaming bound variables (or vice-versa)}$$

Hence the free variable set must be the same (not renamed).

7.2.2 Substitution

$M[new/old]$ means replace free variable *old* with *new* in M

Only free variables can be substituted. Formally we can describe this as:

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

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

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

- For variables, simply check if equal.
- For lambda abstractions, if the old term is bound, cannot go further, else, switch the bound term for some term not free inside, in the substitution, and not the new value replacing.
- For applications, simply substitute into both λ-terms.

Basic Substitution — **Example Question 7.2.1**

$$x[y/x] = y$$
$$y[y/x] = y$$
$$(x\ y)[y/x] = y\ y$$
$$\lambda x \;.\; x\ y[y/x] = \lambda x \;.\; x\ y$$

7.3 Semantics

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

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

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

7.3.1 Multi-Step Reductions

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

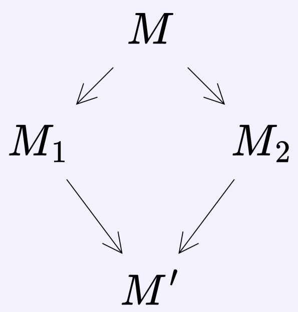
$$\frac{M =_\alpha M'}{M \to_\beta^* M'} \quad \text{(Reflexivity of α-conversion)}$$

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

Confluence — **Definition 7.3.1**

All derivation paths in the derivation tree that reach some normal form, reach the same normal form.

$$\forall M, M_1, M_2.\ [M \to_\beta^* M_1 \wedge M \to_\beta^* M_2 \Rightarrow \exists M'.[M_1 \to_\beta^* M' \wedge M_2 \to_\beta^* M']]$$



β Normal Forms **Definition 7.3.2**

A λ-term is in β-normal form if it contains no *redexes*, and hence cannot be further reduced.

$$\text{is in normal form}(M) \triangleq \forall N.\ M \not\to_\beta N$$

$$\text{has a normal form}(M) \triangleq \exists M'.\ M \to_\beta^* M' \land \text{is in normal form}(M)$$

If a normal form exists, it is unique.

$$\forall M, N_1, N_2, .\ [M \to_\beta^* N_1 \land M \to_\beta^* N_2 \land \text{is-norm-form}(N_1) \land \text{is-norm-form}(N_2) \Rightarrow N_1 =_\alpha N_2]$$

β-equivalence **Definition 7.3.3**

An equivalence relation for $\to_\beta$.

$$M =_\beta N \Leftrightarrow \exists T.\ [M \to_\beta^* T \land N \to_\beta^* T]$$

7.3.2 Reduction Order

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

- Any *redex* in M or N is inside of E
- E is outside of any *redex* in M or N

Innermost Redex **Definition 7.3.4**

A *Redex* with no *redexes* inside of it.

Outermost Redex **Definition 7.3.5**

A *Redex* with no *redexes* outside of it.

We can choose several different orders by which to reduce.

Normal Order **Definition 7.3.6**

- Reduce the *leftmost outermost redex* first.
- This always reduces a λ-term to its normal form if one exists.
- Can perform computations on unevaluated function bodies.
- Not used in any programming languages.

Call By Name **Definition 7.3.7**

- Reduce the *leftmost outermost* first.
- Does not reduce the inside of λ-abstractions.
- Does not always reduce a λ-term to its normal form.
- Passes unevaluated function parameters into function body. Only evaluating a parameter when it is used.
- Used with some variation by haskell, R, and LaTeX.

Call By Values **Definition 7.3.8**

- Reduce the *leftmost innermost redex* first.
- Does not reduce the inside of λ-abstractions.
- Does not always reduce a λ-term to its normal form.
- Evaluate parameters before passing them to function body.
- Terminates less often than *call by name* (e.g if a parameter cannot be normalised, but is never used), but evaluated the parameters only once.
- Used by C, Rust, Java, etc.

η-equivalence **Definition 7.3.9**

Captures equality better than $=_\beta$.

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

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

For example with the basic application of f:

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

7.3.3 Definability

λ-definable **Definition 7.3.10**

Partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is λ-*definable* if and only if there is a *closed* λ-term M where:

$$f(x_1, \dots, x_n) = y \Leftrightarrow M\ \underline{x_1}\ \dots\ \underline{x_n} =_\beta y$$

And

$$f(x_1, \dots, x_n) \uparrow \Leftrightarrow M\ \underline{x_1}\ \dots\ \underline{x_n} \text{ has no } \textit{normal form}$$

λ-*definable* specifies what can be computed by the lambda calculus, and is equivalent to *Register Machine Computable* or *Turing Machine Computable*.

7.4 Encoding Mathematics

7.4.1 Encoding Numbers

We represent natural numbers as *Church Numerals*. These are n repeated applications of some function f.

$$\underline{n} \triangleq \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{n \text{ times}}\dots) \text{ with } n \text{ applications of } f$$

$$\begin{aligned}
\underline{0} &\triangleq \lambda f\ .\ \lambda x\ .\ x \\
\underline{1} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ x \\
\underline{2} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ x \\
\underline{3} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ x \\
\underline{4} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ f\ x \\
\underline{5} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ f\ f\ x \\
&\vdots
\end{aligned}$$

7.4.2 Encoding Addition

Addition is represented as a function application:

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

$$\underline{m+n} \triangleq \underbrace{(\lambda m\ .\ \lambda n\ .\ \lambda f\ .\ \lambda x\ .\ m\ f\ (n\ f\ x))}_{+}\ \underline{m}\ \underline{n}$$

By applying the functions, we have f applied $m+n$ times, representing the *Church Numeral* $\underline{m+n}$.

7.4.3 Encoding Multiplication

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

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

Each application of the f inside m is substituted for n applications of f, using the above λ-abstraction we get $m \times n$ applications of f.

7.4.4 Exponentiation

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

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

7.4.5 Conditional

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

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

If $\underline{m} = \underline{0} = \lambda f \ . \ \lambda x \ . \ x$ then x is returned, which will be x_1.

If not zero, then the f applied returns x_2, so any number of applications of f, results in x_2.

7.4.6 Successor

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

We simply take $\underline{m}$ and apply f one more time

$$\underline{m+1} \triangleq \underbrace{(\lambda m \ . \ \lambda f \ . \ \lambda x \ . \ f \ (m \ f \ x))}_{\text{succ}} \ \underline{m}$$

7.4.7 Pairs

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

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

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

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

Q2bv - 2020/21 **Exam Question 7.4.1**

... continued from 2bvi - 2020/21

Using Church numerals, give an equivalent λ-term (program) C, i.e. for all $n > 0$ we have $C \ \underline{n} \to_\beta^* \underline{m}$ if and only if the execution of register machine K also halts with $R_0 = m$ when started with $R_0 = n$.

You can use the pre-defined operations from the lecture ($plus$, $mult$, $succ$, $pred$, ifz, etc.) and also integer division (div) and reminder (rem). It helps to use various subroutines.

Q2d - 2021/22 **Exam Question 7.4.2**

Consider the following recursively defined sequence of integers x_i:

$$x_0 = x_1 = 1$$

$$x_i = x_{i-2}^2 + 2x_{i-1}$$

Implement this in the λ-calculus using Church numerals, i.e. write a lambda term f such that $f \ \underline{n}$ reduces to $\underline{x_n}$.

You can use functions defined in the lecture, e.g $plus$, $mult$, ifz etc. It might help to define subroutines.

Sketch the execution of $f\ \underline{2}$ (you can use $\to_\beta^*$ rather than $\to_\beta$).

7.4.8 Predecessor

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

We cannot remove applications of f, however we can use a pair to count up until the successor is $\underline{m}$.

Hence we first need a function to get the next pair from the current:

$$transition\ p \triangleq \underbrace{(\lambda n\ .\ newpair\ (second\ n)\ ((second\ n) + 1))}_{\text{transition function}}\ p$$

We can then simply run the transition n times on a pair starting by using $f = transition$ and $x = newpair\ \underline{0}\ \underline{0}$.

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

$$pred(n) \triangleq \underbrace{(\lambda n\ .\ n\ transition\ (newpair\ \underline{0}\ \underline{0})\ first)}_{\text{predecessor}}\ \underline{n}$$

A simpler definition of predecessor is:

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

7.4.9 Subtraction

We can use the predecessor function for subtraction. By applying the predecessor function $\underline{n}$ times on some number $\underline{m}$ we get $\underline{m - n}$.

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

7.5 Combinators

Combinator **Definition 7.5.1**

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

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

Only SKI are required to define any *computable function* (can remove even λ-abstraction, this is called *SKI-Combinator Calculus*).

The Y-Combinator is used for recursion. In one step of β-reduction:

$$Y\ f \to_\beta f\ (Y\ f)$$

We cannot define λ-terms in terms of themselves, as the λ-term is not yet defined, and infinitely large λ-terms are not allowed.

We can use the $Y - Combinator$ to create recursion in the absence of recursive λ-term definitions.

Fixed-Point Combinator **Definition 7.5.2**

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

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

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

Factorial **Example Question 7.5.1**

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

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

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

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

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

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

Q2c - 2020/21 **Exam Question 7.5.1**

Consider the term $\mathbf{Y}' = \mathbf{UU}$ with $\mathbf{U} = \lambda u\ x\ .\ x\ uux$.

Why is $\mathbf{Y}$ (and $\mathbf{U}$) a combinator? Show that it is an alternative implementation of the *fixed-point combinator*.

Chapter 8

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.

Register Machine Sim   Linked simulator was developed by Yìtáng Chén.

These notes were written by Oliver Killane.