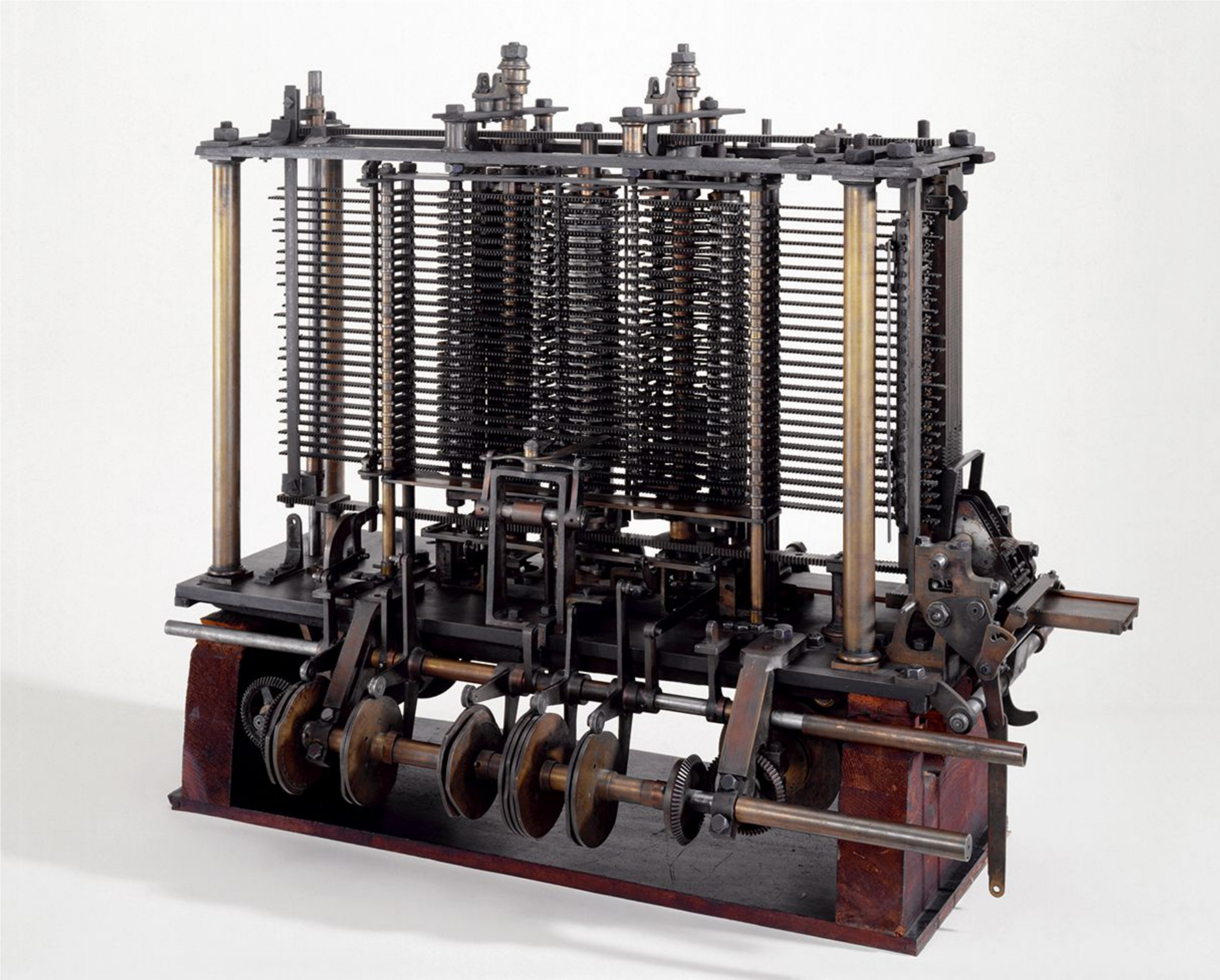


50003

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]
```

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

Al-Khwarizmi *Extra Fun!* 1.2.3

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

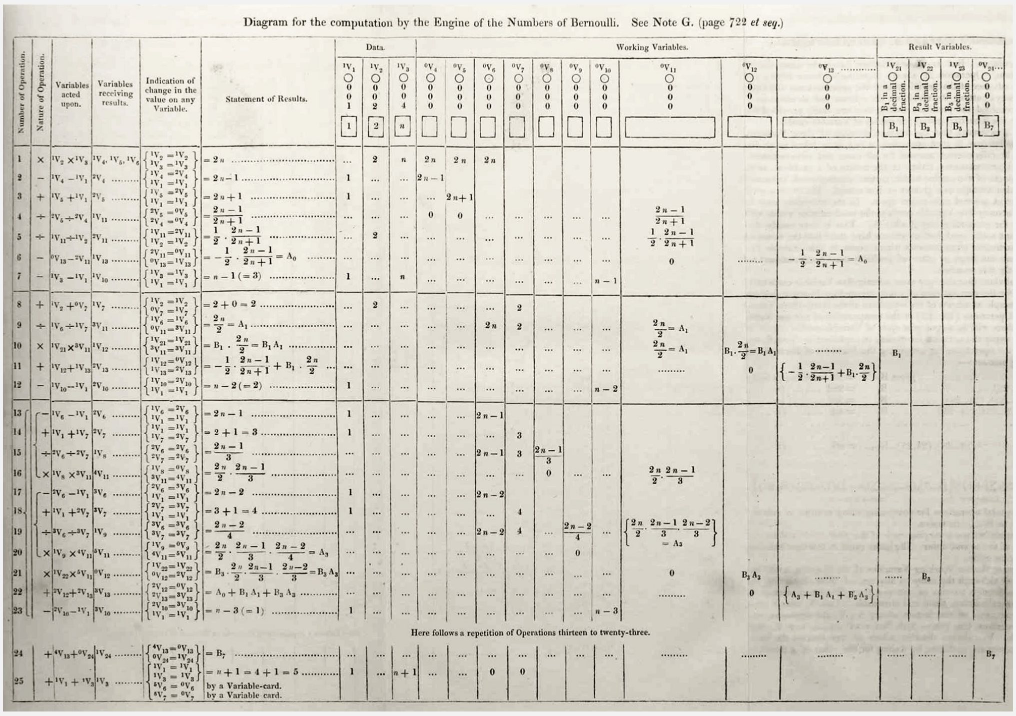
His book title الجبر *"al-jabr"* resulted in the word *algebra* and subsequently algorithm.

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

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

Note G *Extra Fun!* 1.2.4

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



Babbage's Machines *Extra Fun!* 1.2.5

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

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

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

1.3 Decision Problems

Formulas **Definition 1.3.1**

Well formed logical statements that are a sequence of symbols form a given formal language. e.g $(p \lor q) \land i$ 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 porperty of a formula that it has a proof).

The associated decision procedure is:

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

1.3.1 Hilbert's Entscheidungsproblem

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

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

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

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

'*Entscheidungsproblem*' means 'decision problem'

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

1.4 Algorithms

1.4.1 Algorithms Informally

Common features of Algorithms:

Finite	Description of the procedure in terms of elementary operations.
Deterministic	If there is a next step, it is uniquely determined - that is on the same data, the same steps will be made.
Terminate?	Procedure may not terminate on some input data, however we can recognize when it terminates and what the result is.

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

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

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

Algorithms Formalised

Any formal definition of an algorithm should be:

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

1.4.2 The Halting Problem

The *Halting problem* is a *decision problem* with:

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

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

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

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

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

1.4.3 Algorithms as Functions

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

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

Domains **Definition 1.4.1**

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

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

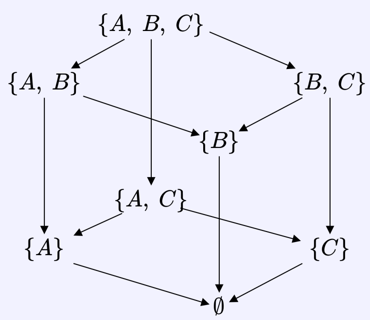


Diagram of $\subseteq$ for sets $\subseteq \{A, B, C\}$

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

1.4.4 Haskell Programs

Example using a basic implementation of power.

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

```
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)$\hat{2}$
⇝ 7 * ((power' 7 1)$\hat{2}$)$\hat{2}$
⇝ 7 * ((7 * (power' 7 0)$\hat{2}$)$\hat{2}$)$\hat{2}$
⇝ 7 * ((7 * (1)$\hat{2}$)$\hat{2}$)$\hat{2}$
⇝ 16807

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

1.5 Program Semantics

Denotational Semantics — **Definition 1.5.1**

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

Operational Semantics — **Definition 1.5.2**

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

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

Chapter 2

While Language

2.1 SimpleExp

We can define a simple expression language (*SimpleExp*) to work on:
$$E \in \mathit{SimpleExp} \ ::= \ n \mid E + E \mid E \times E \mid \ \dots$$

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**

How could we break the totality of SimpleExp?

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

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

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

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

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

We can hence create the derivation:

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

2.1.2 Small Step Semantics

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

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

Rules

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

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

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

Normal Form **Definition 2.1.5**

E is in its normal form (irreducible) 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**

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

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

Determinacy **Definition 2.1.7**

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

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**

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

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

Add a rule to break determinacy without breaking confluence.

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

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

2.2 While

2.2.1 Syntax

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

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

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

2.2.2 States

Partial Function — **Definition 2.2.1**

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

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

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

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

For example:

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

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

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

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

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

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

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

(s' equals s where x maps to a)

2.2.3 Rules

Expressions

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

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

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

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

Booleans

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

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

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

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

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

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

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

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

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

Assignment

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

Sequential Composition

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

Conditionals

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

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

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

While

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

2.2.4 Properties

The execution relation ($\to_c$) 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 $\rightarrow_b$ and $\rightarrow_e$ are normalising, but $\rightarrow_c$ is not as it may not have a normal form.

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

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

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

2.2.7 Side Effecting Expressions

If we allow programs such as:

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

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

(value depends on evaluation order)

2.2.8 Short Circuit Semantics

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

2.2.9 Strictness

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

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

2.2.10 Complex Programs

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

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

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

Execute! **Example Question 2.2.1**

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

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

Start

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

Get x variable

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

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

Result:

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

Assign to y variable

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

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

Result:

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

Eliminate skip

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

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

Result:

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

Assign a

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

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

Result:

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

Eliminate skip

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

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

Result:

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

Expand while

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

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

Result:

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

Get y variable

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

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

Result:

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

Complete if boolean

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

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

Result:

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

Evaluate if

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

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

Result:

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

Evaluate Expression a

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

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

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

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

Result:

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

Eliminate Skip

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

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

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

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

Result:

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

Assign 2 to y

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

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

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 $s = (x \mapsto 3, y \mapsto 2, z \mapsto 42, a \mapsto 3)$:

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

Result:

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

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 $plus(zero, zero) = zero$

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

Inductive Case

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

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

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*:

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

P **Example Question 3.1.1**

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

UNFINISHED!!!

3.2 Induction over SimpleExp

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

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

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

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

3.2.1 Many Steps of Evaluation

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

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

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

3.2.2 Multi-Step Reductions

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

Base Case

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

Inductive Case

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

Definition

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

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

Properties of $\rightarrow$

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

3.2.3 Confluence of Small Step

We can prove a lemma expressing confluence:

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

Lemma ⇒ Confluence

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

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

3.2.4 Determinacy of Small Step

We create a property P:

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

There are 3 rules that apply:

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

Base Case

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

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

Hence $P(n)$

Inductive Step

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

$$IH_1 = P(E_1)$$

$$IH_2 = P(E_2)$$

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

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

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

For B:

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

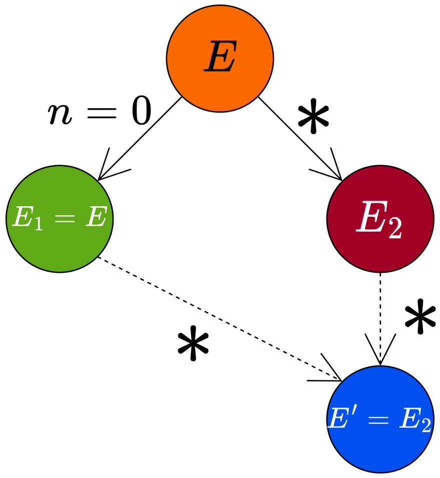
For C:

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

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

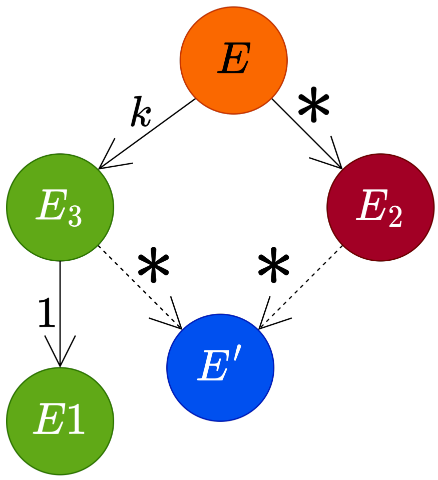
Base Case

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

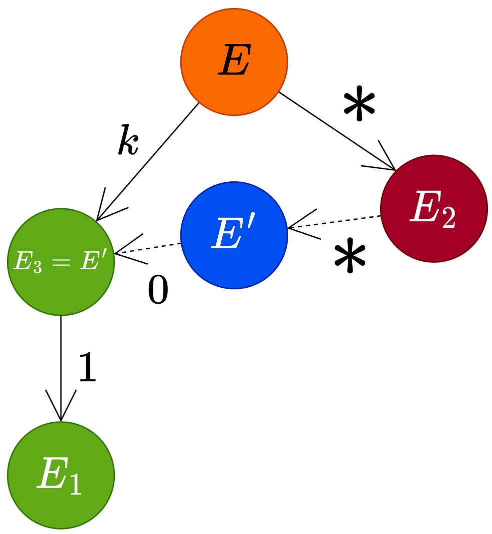


Inductive Case

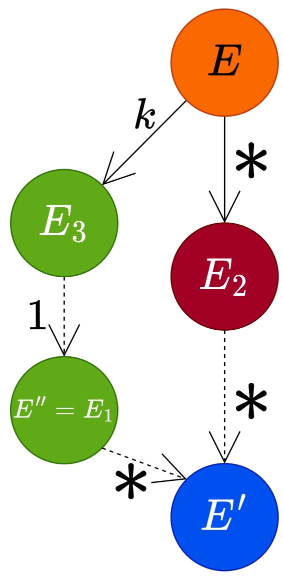
Next we assume confluence for up to k steps, and attempt to prove for $k + 1$ steps.



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



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



3.3 Multi-Step Reductions

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

Lemma — **Definition 3.3.1**

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

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

Corollary — **Definition 3.3.2**

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

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

3.3.1 Lemmas

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

3.3.2 Corollaries

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

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

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

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

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

Base Case

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

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

Inductive Step

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

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

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

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

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

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

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

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

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

3.3.4 Multi-Step Reductions

Lemmas

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

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

Base Case

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

Inductive Step

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

Proof of the Lemma

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

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

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

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

3.3.5 Determinacy of $\rightarrow$ for Exp

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

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

Determinacy:

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

We prove this using property P:

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

Base Case: $E = x$

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

(1)	take $E_1 \in \mathit{Exp}$, $s_1 \in \mathit{State}$ such that $\langle n, s\rangle \rightarrow \langle E_1, s_1\rangle$	(Initial setup)
(2)	take $E_2 \in \mathit{Exp}$, $s_2 \in \mathit{State}$ such that $\langle n, s\rangle \rightarrow \langle E_2, s_2\rangle$	(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 $\langle x, s\rangle \rightarrow \langle E_1, s_1\rangle$	(Initial setup)
(2)	take $E_2 \in \mathbb{N}$, $s_2 \in \mathit{State}$ such that $\langle x, s\rangle \rightarrow \langle E_2, s_2\rangle$	(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

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

Determinacy

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

Confluence

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

Unique Answer

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

No Normalisation

There exist derivations of infinite length for while.

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

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

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

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

Chapter 4

Credit

Image Credit

Front Cover   Analytical Engine - Science Museum London

Content

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

These notes were written by Oliver Killane.