Computable Functions

In this part of the course, we study several formal definitions of algorithm. These definitions are equivalent, providing different ways of describing the notion of **computable function**.

Slide 1

Algorithms, informally

People tried to find an algorithm to solve Hilbert's Entscheidungsproblem, without success.

A natural question was then to ask whether it was possible to **prove** that such an algorithm did not exist. To ask this question properly, it was necessary to provide a **formal** definition of algorithm.

Common features of the (historical) examples of algorithms:

- **finite** description of the procedure in terms of elementary operations;
- **deterministic**, next step is uniquely determined if there is one;
- procedure may not terminate on some input data, but we can recognise when it does terminate and **what** the **result** will be.

So far, we have studied the operational and denotational semantics of a simple while language, *While*. The denotational semantics of *While* gives mathematical meaning to while programs as special functions (actually, the so-called continuous functions) over special sets (domains). We have given an indication of this world of denotational semantics, by providing an interpretation (meaning) of while programs as state transformers $\mathtt{ST} = [\Sigma \to \Sigma_\perp]$ where Σ is the set of all states. However, although the denotational semantics provides a *fundamental* mathematical meaning to programs, it does not provide us with a direct definition of what it means to compute something. Meanwhile, the operational semantics of *While* does provide us with a formal description of what it means to compute something. Given an initial state, we know precisely how a while program computes: it either computes for ever, gets stuck, or terminates yielding a final state (the result). Although the commands in *While* are important programming constructs, the language does not seem right as a *fundamental* definition of algorithm. Instead, we

will initially explore definitions of algorithm which are closer to how machines compute. We first give a formal definition of *register machine*, which provides a simple description of a computing machine. We then give a definition of *Turing machine*, which is more complicated description of a computing machine. Although more complicated, every computer scientist should know about Turing machines! Finally, we give a definition of Church's lambda-calculus. This provides a completely different way of describing computation which is much nearer to the notion of function rather than the notion of computing machine. What is amazing is that these different definitions are actually equivalent.
In the Compilers course, you have seen the translation of a while language into register machines. Each course uses slightly different definitions, because the different courses are emphasising different points. Despite this, ideas from each course transfer. Can you define a translation from *While* to the register machines described in this course?

Slide 2

Algorithms as Special Functions

Turing and Church's equivalent definitions of algorithm capture the notion of **computable function**: an algorithm expects some input, does some calculation and, if it terminates, returns a unique result.

We first study **register machines**, which provide a simple definition of algorithm. We describe the **universal register machine** and introduce the **halting problem**, which is probably the most famous example of a problem that is not computable.

We then move to **Turing machines** and **Church's λ-calculus**.

Register Machines

The register machine gets its name from its one or more uniquely addressed *registers*, each of which holds a natural number. There are several versions of register machines. We are using the Minski register machines. The work

on register machines occurred in the 1950s and 1960s. One motivation was that people were trying to show that Hilbert's 10th problem on Diophantine equations was undecidable. This was finally cracked in 1970s using Turing machines, and a simpler proof was given in the 1980s using register machines. Register machines are (apparently) particularly suited to constructing Diophantine equations [Matiyasevich].

Slide 3

Register Machines, informally

Register machines operate on natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ stored in (idealized) registers using the following "elementary operations":

- add 1 to the contents of a register
- test whether the contents of a register is 0
- subtract 1 from the contents of a register if it is non-zero
- jumps ("goto")
- conditionals ("if_then_else_")

Slide 4

Register Machines

Definition

A **register machine** (sometimes abbreviated to RM) is specified by:

- finitely many **registers** $R_0, R_1, \ldots, R_n$, each capable of storing a natural number;
- a **program** consisting of a finite list of instructions of the form $label : body$ where, for $i = 0, 1, 2, \ldots$, the $(i+1)^{\text{th}}$ instruction has label L_i. The instruction **body** takes the form:

$R^+ \to L'$	add 1 to contents of register R and jump to instruction labelled L'
$R^- \to L', L''$	if contents of R is > 0, then subtract 1 and jump to L', else jump to L''
$HALT$	stop executing instructions

Slide 5

Example

Registers

$R_0 \ R_1 \ R_2$

Program

$L_0 : R_1^- \to L_1, L_2$

$L_1 : R_0^+ \to L_0$

$L_2 : R_2^- \to L_3, L_4$

$L_3 : R_0^+ \to L_2$

$L_4 : HALT$

Example Computation

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

Exercise Consider the following program, acting on registers R_0, R_1, R_2, R_3:

Program

$L_0 : R_1^- \to L_1, L_6$
$L_1 : R_2^- \to L_2, L_4$
$L_2 : R_0^+ \to L_3$
$L_3 : R_3^+ \to L_1$
$L_4 : R_3^- \to L_5, L_0$
$L_5 : R_2^+ \to L_4$
$L_6 : HALT$

Give the example computation starting from initial configuration $(0, 2, 3, 0)$.

Slide 6

Register Machine Configuration

A register machine **configuration** has the form:

$$c = (\ell, r_0, \ldots, r_n)$$

where ℓ = current label and r_i = current contents of R_i.

Notation "$R_i = x$ [in configuration c]" means $c = (\ell, r_0, \ldots, r_n)$ with $r_i = x$.

Initial configurations

$$c_0 = (0, r_0, \ldots, r_n)$$

where r_i = initial contents of register R_i.

Slide 7

Register Machine Computation

A **computation** of a RM is a (finite or infinite) sequence of configurations

$$c_0, c_1, c_2, \ldots$$

where

- $c_0 = (0, r_0, \ldots, r_n)$ is an initial configuration;
- each $c = (\ell, r_0, \ldots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by carrying out the program instruction labelled L_ℓ with registers containing $r_0, \ldots, r_n$.

Slide 8

Halting Computations

For a finite computation $c_0, c_1, \ldots, c_m$, the last configuration $c_m = (\ell, r, \ldots)$ is a **halting** configuration: that is, the instruction labelled L_ℓ is

either $HALT$ (a ' proper halt')

or $R^+ \to L$, or $R^- \to L, L'$ with $R > 0$, or $R^- \to L', L$ with $R = 0$ and there is no instruction labelled L in the program (an 'erroneous halt')

For example, the program

$$\begin{array}{l} L_0 : R_1^+ \to L_2 \\ L_1 : HALT \end{array}$$

halts erroneously.

Notice that it is always possible to modify programs (without affecting their computations) to turn all erroneous halts into proper halts by adding extra $HALT$ instructions to the list with appropriate labels.

Slide 9

Non-halting Computations

There are computations which never halt. For example, the program

$$\begin{array}{l} L_0 : R_1^+ \to L_0 \\ L_1 : HALT \end{array}$$

only has infinite computation sequences

$$(0, r), (0, r+1), (0, r+2), \ldots$$

Slide 10

Graphical representation

- One node in the graph for each instruction $label : body$, with the node labelled by the register of the instruction body; notation $[L]$ denotes the register of the body of label L
- Arcs represent jumps between instructions
- Initial instruction $START$.

Instruction	Representation
$R^+ \to L$	$R^+ \longrightarrow [L]$
$R^- \to L, L'$	R^- → $[L]$, R^- → $[L']$
$HALT$	$HALT$
L_0	$START \longrightarrow [L_0]$

Slide 11

Example

Registers

$R_0\ R_1\ R_2$

Program

$L_0 : R_1^- \to L_1, L_2$

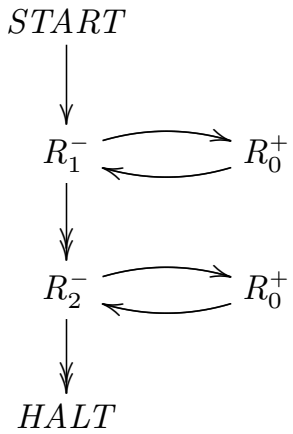
$L_1 : R_0^+ \to L_0$

$L_2 : R_2^- \to L_3, L_4$

$L_3 : R_0^+ \to L_2$

$L_4 : HALT$

Graphical Representation



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$.

The graphical representation is a bit confusing. There is one node in the graph for each instruction $label : body$. However, the nodes are only labelled with the registers of the instruction bodies. For example, in slide 10, we have two nodes labelled R_0^+. The top node corresponds to the instruction $L_1 : R_0^+ \to L_0$, and the bottom node to $L_3 : R_0^+ \to L_2$. The initial instruction $START$ is essential, as the graphical representation looses the sequential ordering of instructions.

Exercise Recall the following program acting on registers R_0, R_1, R_2, R_3:

Program

$L_0 : R_1^- \to L_1, L_6$
$L_1 : R_2^- \to L_2, L_4$
$L_2 : R_0^+ \to L_3$
$L_3 : R_3^+ \to L_1$
$L_4 : R_3^- \to L_5, L_0$
$L_5 : R_2^+ \to L_4$
$L_6 : HALT$

What is the graphical representation of this program?

Slide 12

Partial functions

Register machine computation is **deterministic**: in any non-halting configuration, the next configuration is uniquely determined by the program.

So the relation between initial and final register contents defined by a register machine program is a **partial function**...

Definition A **partial function** from a set X to a set Y is specified by any subset $f \subseteq X \times Y$ satisfying

$$(x, y) \in f \text{ and } (x, y') \in f \text{ implies } y = y'.$$

Slide 13

Partial Functions

Notation

- "$f(x) = y$" means $(x, y) \in f$
- "$f(x)\downarrow$" means $\exists y \in Y\,(f(x) = y)$
- "$f(x)\uparrow$" means $\neg\exists y \in Y\,(f(x) = y)$
- $X \rightharpoonup Y$ = set of all partial functions from X to Y
  $X \to Y$ = set of all (**total**) functions from X to Y

Definition. A **partial function** from a set X to a set Y is **total** if it satisfies

$$f(x)\downarrow$$

for all $x \in X$.

Slide 14

Computable functions

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

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

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

The *I/O convention* is somewhat arbitrary: in the initial configuration, registers $R_1, \ldots, R_n$ store the function's arguments (with all others zeroed); in the halting configuration, register R_0 stores it's value (if any). Notice that there may be many different register machines that compute the same partial function f.

Slide 15

Example

Registers

$R_0\ R_1\ R_2$

Program

$L_0 : R_1^- \to L_1, L_2$

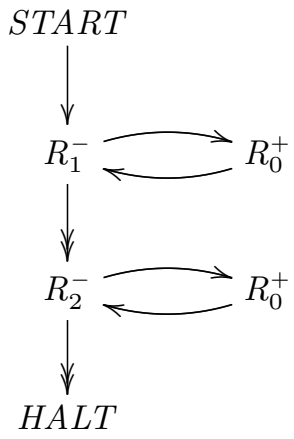
$L_1 : R_0^+ \to L_0$

$L_2 : R_2^- \to L_3, L_4$

$L_3 : R_0^+ \to L_2$

$L_4 : HALT$

Graphical Representation

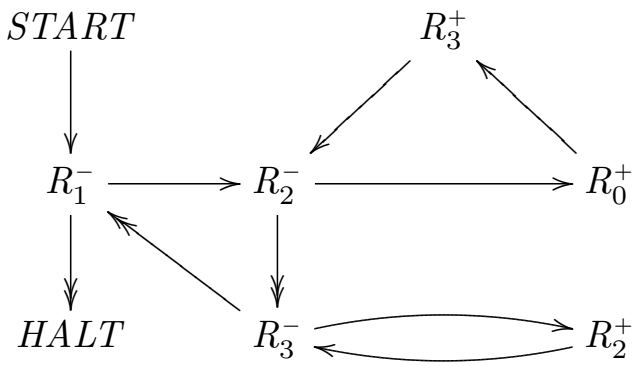


If the machine starts with registers $(R_0, R_1, R_2) = (0, x, y)$, then it halts with registers $(R_0, R_1, R_2) = (x + y, 0, 0)$.

The notation is a little confusing. The slide states that, if the machine starts with registers $(R_0, R_1, R_2) = (0, x, y)$, then it halts with registers $(R_0, R_1, R_2) = (x + y, 0, 0)$. This description focusses on registers, and demonstrates that $f(x, y) \triangleq x + y$ is computable. (The notation $f(x, y) \triangleq x + y$ means that $f(x, y)$ 'is defined to be equal to' $x + y$.) Compare this description with the description using configurations in slide 11: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$. This description also gives information about the initial and final labels. The configuration $(0, 0, x, y)$ means that the first component is the initial label 0, the second component is initially set to zero and will eventually give the final answer when the computation halts, and the third and fourth components provide the two input values of the function. From configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$.

Slide 16

Multiplication $f(x,y) \triangleq xy$ is computable



If the machine starts with registers $(R_0, R_1, R_2, R_3) = (0, x, y, 0)$, then it halts with registers $(R_0, R_1, R_2, R_3) = (xy, 0, y, 0)$.

Exercise Construct a register machine that computes the function $f(x,y) \triangleq x + y$.

The following arithmetic functions are all computable. The proofs are left as exercises.

1. Projection: $p(x,y) \triangleq x$
2. Constant: $c(x) \triangleq n$
3. Truncated subtraction: $x \dot{-} y \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$
4. Integer division: $x \; div \; y \triangleq \begin{cases} \text{integer part of } x/y & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$
5. Integer remainder: $x \; mod \; y \triangleq x \dot{-} y(x \; div \; y)$
6. Exponentiation base 2: $e(x) \triangleq 2^x$
7. Logarithm base 2: $\log_2(x) \triangleq \begin{cases} \text{greatest } y \text{ such that } 2^y \leq x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$

Coding Programs as Numbers

So far, we have only seen how to write simple arithmetical operations as register-machine programs. The Turing/Church solution of the Entscheidungsproblem and the Halting problem uses the fundamentally important idea that (formal descriptions of) algorithms can be the data on which algorithms act. Recall the following slide from lecture 1.

Slide 17

The Halting Problem

The Halting Problem is the decision problem with

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

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

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

To realise this idea of algorithms being used as input data in the context of Register Machines, we have to be able to code register-machine programs as numbers. (In general, such codings are often called **Gödel numberings**.) To do this, first we have to code pairs of numbers and lists of numbers as numbers. There are many ways to do this. We fix upon one way.

Slide 18

Numerical Coding of Pairs

Definition

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

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

Result

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

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

Recall the definition of bijection from discrete maths.

The notation $0\mathrm{b}11011$ is sometimes used to emphasise that the number, in this case 11011, is in binary. We will also use the notation $0\mathrm{b}x$ for $x \in \mathbb{N}$ to denote the binary number of x. We investigate a few examples of $\langle\!\langle x, y \rangle\!\rangle$ for small examples of x and y:

$$\begin{array}{llll} \langle\!\langle 0,0 \rangle\!\rangle = 1 & \langle\!\langle 1,0 \rangle\!\rangle = 2 & \langle\!\langle 2,0 \rangle\!\rangle = 4 & \langle\!\langle 3,0 \rangle\!\rangle = 8 \\ \langle\!\langle 0,1 \rangle\!\rangle = 3 & \langle\!\langle 1,1 \rangle\!\rangle = 6 & \langle\!\langle 2,1 \rangle\!\rangle = 12 & \ldots \\ \langle\!\langle 0,2 \rangle\!\rangle = 5 & \langle\!\langle 1,2 \rangle\!\rangle = 10 & \langle\!\langle 2,2 \rangle\!\rangle = 20 & \ldots \\ \langle\!\langle 0,3 \rangle\!\rangle = 7 & \ldots & & \end{array}$$

Slide 19

Numerical Coding of Pairs

Definition

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

Sketch Proof of Result

It is enough to observe that

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

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

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

To show that

$$\boxed{0\mathrm{b}\langle\!\langle x, y \rangle\!\rangle} = \boxed{0\mathrm{b}y \mid 1 \mid \underbrace{0 \cdots 0}_{x\ 0s}}$$

observe that $\langle\!\langle x, y \rangle\!\rangle \triangleq 2^x(2y+1) = 2^{x+1}y + 2^x$.
To show $\langle\!\langle -, - \rangle\!\rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$ is one-to-one, assume that $\langle\!\langle x_1, y_1 \rangle\!\rangle = \langle\!\langle x_2, y_2 \rangle\!\rangle$, and either $x_1 \neq x_2$ or $y_1 \neq y_2$ or both. Since $\langle\!\langle x_1, y_1 \rangle\!\rangle = \langle\!\langle x_2, y_2 \rangle\!\rangle$, we have $0\mathrm{b}\langle\!\langle x_1, y_1 \rangle\!\rangle = 0\mathrm{b}\langle\!\langle x_2, y_2 \rangle\!\rangle$ and hence

$$\boxed{0\mathrm{b}y_1 \mid 1 \mid \underbrace{0 \cdots 0}_{x_1\ 0s}} = \boxed{0\mathrm{b}y_2 \mid 1 \mid \underbrace{0 \cdots 0}_{x_2\ 0s}}$$

which cannot hold as either $x_1 \neq x_2$ or $y_1 \neq y_2$ or both.

To show $\langle\!\langle -, - \rangle\!\rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$ is onto, assume not. We know that $\langle\!\langle 0, 0 \rangle\!\rangle = 1$. Hence, there must be a smallest $n \in \mathbb{N}^+$ such that $n = \langle\!\langle u, v \rangle\!\rangle$ for some $u, v \in \mathbb{N}$ and $n + 1 \neq \langle\!\langle x, y \rangle\!\rangle$ for all $x, y \in \mathbb{N}$. So, $n = 0\mathrm{b}v\,1\ \underbrace{0 \cdots 0}_{u\ 0s}$ and $0\mathrm{b}(n+1) = 0\mathrm{b}(n) + 1 = 0\mathrm{b}v\,1\ \underbrace{0 \cdots 0}_{u\ 0s} + 1$.
If $u \neq 0$, then $n + 1 = \langle\!\langle 0, w \rangle\!\rangle$ where $0\mathrm{b}w = 0\mathrm{b}v\,1\ \underbrace{0 \cdots 0}_{u-1\ 0s}$. If $u = 0$,

then $n+1 = \langle\!\langle x, y \rangle\!\rangle$ where x is one plus the number of zeros before the first one in $0\mathtt{b}(v+1)$ and y is the (possibly zero) binary number after the first one.
Here's another proof! To prove $\langle\!\langle -, - \rangle\!\rangle$ is one-to-one, assume $\langle\!\langle x_1, y_1 \rangle\!\rangle = \langle\!\langle x_2, y_x \rangle\!\rangle$: that is,

$$2^{x_1}(2y_1+1) = 2^{x_2}(2y_2+1)$$

If $x_1 > x_2$, then $2^{x_1-x_2}(2y_1+1) = 2y_2+1$ which is impossible. A similar argument shows that $x_1 < x_2$ is impossible. Hence, $x_1 = x_2$ and $2y_1+1 = 2y_2+1$, which implies that $y_1 = y_2$ and $\langle\!\langle -, - \rangle\!\rangle$ is one-to-one.
To prove $\langle\!\langle -, - \rangle\!\rangle$ is onto, assume for contradiction that there is a smallest $n \in \mathbb{N}$ such that there is no $x, y \in \mathbb{N}$ with $\langle\!\langle x, y \rangle\!\rangle = n$. If n is even, then $n = 2m$ with $m < n$. Hence, $m = 2^{x_1}(2y_1+1)$ for some $x_1, y_1 \in \mathbb{N}$. Then $n = 2^{x_1+1}(2y_1+1)$. If n is odd, then $n = 2m+1$, $m < n$ and $m = 2^{x_1}(2y_1+1)$ for some $x_1, y_1 \in \mathbb{N}$. If $x_1 = 0$, then $n = 2(2y_1+1)$. If $x_1 \geq 1$, then $n = 2.(2^{x_1}(2y_1+1)+1) = 2(2(2^{x_1}y_1 + 2^{x_1-1})+1)$. Hence, $\langle\!\langle -, - \rangle\!\rangle$ is onto.

Slide 20

Numerical Coding of Lists

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

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

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

Slide 21

Numerical Coding of Lists

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

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

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

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

Slide 22

Numerical Coding of Lists

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

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

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

Examples

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

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

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

Slide 23

Numerical Coding of Lists

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

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

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

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

Slide 24

Numerical Coding of Lists

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

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

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

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

Sketch Proof

The proof follows by observing that

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

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

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

Inductive step Assume

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

By the definitions, we have

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

The induction hypothesis now gives the result. Using this result, ${}^{\ulcorner}L^{\urcorner}$ is clearly one-to-one and onto. To convince yourself of this, choose a few binary numbers n and give the corresponding list L_n such that $0\mathsf{b}^{\ulcorner}L_n{}^{\urcorner} = n$.

Slide 25

Numerical Coding of Programs

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

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

where the numerical code ${}^{\ulcorner}body^{\urcorner}$ of an instruction body is defined by:

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

Since $\langle\!\langle -, - \rangle\!\rangle : \mathbb{N}\times\mathbb{N} \to \mathbb{N}^+$, $\langle -, - \rangle : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ and ${}^{\ulcorner}-{}^{\urcorner} : List\,\mathbb{N} \to \mathbb{N}$ are bijections, the functions ${}^{\ulcorner}-{}^{\urcorner}$ from bodies to natural numbers and ${}^{\ulcorner}-{}^{\urcorner}$ from RM programs to $\mathbb{N}$ are bijections.

In the next section, we will introduce the *Universal Register Machine*. The Universal Register Machine carries out the following computation:

> starting with $R_0 = 0$, $R_1 = e$ (the code of the program), $R_2 = a$ (code of the list of arguments), and all other registers zeroed:
>
> - decode e as a RM program P
> - decode a as a list of register values $a_1, \ldots, a_n$
> - carry out the computation of the RM program P starting with $R_0 = 0, R_1 = a_1, \ldots, R_n = a_n$ (and any other registers occurring in P set to 0).

It is therefore important for you to understand what it means for a number $x \in \mathbb{N}$ to decode to a unique instruction $body(x)$, and for a number $e \in \mathbb{N}$ to decode to a unique program $prog(e)$.

Slide 26

Decoding Numbers as Bodies and Programs

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

if $x = 0$ then $body(x)$ is $HALT$,
else ($x > 0$ and) let $x = \langle\!\langle y, z \rangle\!\rangle$ in
if $y = 2i$ is even, then $body(x)$ is $R_i^+ \to L_z$,
else $y = 2i + 1$ is odd, let $z = \langle j, k \rangle$ in
$body(x)$ is $R_i^- \to L_j, L_k$

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

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

Slide 27

Example of $prog(e)$

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

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

Notice that, when $e = 0$, we have $0 = \ulcorner[]\urcorner$ so $prog(0)$ is the program with an empty list of instructions, which by convention we regard as a RM that does nothing (i.e. that halts immediately). Also, notice in slide 26 the jump to a label with no body (an erroneous halt). Again, choose some numbers and see what the register-machine programs they correspond to.