CS_275 Automata and Formal Language Theory

Course Notes
Part III: Limits of Computation
Chapt. III.3: Turing Machines

Anton Setzer

http://www.cs.swan.ac.uk/∼csetzer/lectures/automataFormalLanguage/current/index.html

April 29, 2018

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

(a) Definition of the Turing Machine

- There are two problems with the model of a URM:
- Execution of a single URM instruction might take arbitrarily long:
  - Consider instruction $\mathrm{R}_n := \mathrm{R}_n + 1$.
  - If R_n contains in binary $\underbrace{111\cdots111}_{k \text{ times}}$, this instruction replaces it by $1\underbrace{000\cdots000}_{k \text{ times}}$.
  - We have to replace k symbols 1 by 0.
  - k is arbitrary
    $\rightarrow$ this single step might take arbitrarily long time.

First Problem of URMs

- That incrementing a number by one takes arbitrarily many steps happens on a real computer as well:
  - If we want to represent arbitrary big numbers on the computer, we have to represent them by multiple machine integers
    - Then incrementing a number by one will correspond to arbitrarily many machine instructions (although usually only a few).
  - However, often in complexity theory this problem is ignored because the effect is marginal in real applications.
    - The exception are applications in which very big integers occur, e.g. tests for primality. There this effect cannot be ignored any more.

First Problem of URMs

  - If one takes this effect into account, one needs in many examples to multiply the running time by a factor of $\ln(n)$, where n is the largest number occurring.
- Therefore URMs unsuitable as a basis for defining the precise complexity of algorithms.
- However, there are theorems linking complexity of URMs to actual complexities of algorithms.

Second Problem of URMs

- We aim at a notion of computability, which covers all possible ways of computing something, independently of any concrete machine.
- URMs are a model of computation which covers current standard computers.
- However, there might be completely different notions of computability, based on symbolic manipulations of a sequence of characters, where it might be more complicated to see directly that all such computations can be simulated by a URM.
- It is more easy to see that such notions are covered by the Turing machine model of computation.

Idea of a Turing Machine

- Idea of a Turing machine (**TM**):
  Analysis of a computation carried out by a human being (**agent**) on a piece of paper.

Idea of a Turing Machine

- Steps in this formulation:
  - Algorithm should be deterministic.
    → The agent will use only finitely many symbols, put at discrete positions on the paper.

		1	5	.	1	6	=				
									1	5	
										9	0
									–	–	–
									2	4	0

Idea of a Turing Machine

- We can replace a two-dimensional piece of paper by one potentially infinite tape, by using a special symbol for a line break.
- Each entry on this tape is called a **cell**:

···	1	5	.	1	6	=				CR								1	5		CR	···

Steps in Formalising TMs

- In the real situation, agent can look at several cells at the same time, but bounded by his physical capability.
  Can be simulated by looking at one cell only at any time, and moving around in order to get information about neighbouring cells.

···	1	5	.	1	6	=				CR								1	5		CR	···
				↑																		
				Head																		

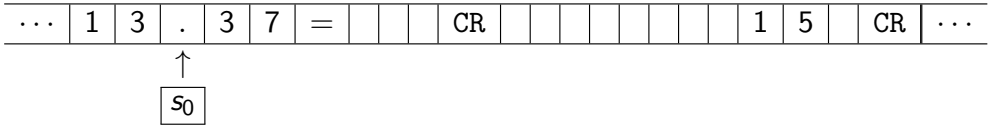
Steps in Formalising TMs

- In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent.
  Each such jump can be simulated by finitely many one-step jumps.
  → Restriction to one-step movements.

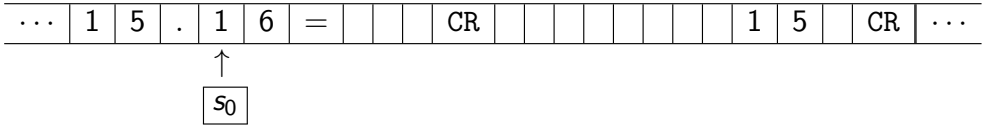
···	1	5	.	1	6	=				CR								1	5		CR	···
			↑																			
			Head																			

Steps in Formalising TMs

- Agent operates purely mechanistically:
  Reads a symbol, and depending on it changes it and makes a movement.
  Agent himself will have only finite memory.
  → There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.

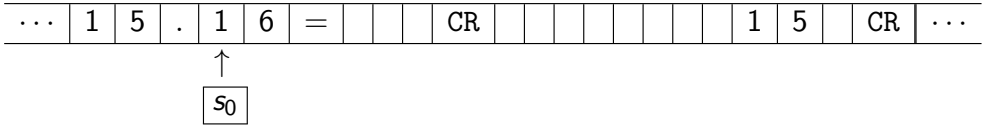


Definition of TMs



- A **Turing machine** is given by the following 5 components:
  - A finite set of symbols Σs, called the **alphabet** of the Turing machine. On the tape, the symbols in Σ will be written.
    - Σ is the Greek capital letter "Sigma".
  - A finite set of states S.

Definition of TMs



- A finite set I consisting of instructions $s \xrightarrow{a/a',D} s'$, where
  - $s, s' \in S$,
  - $a, a' \in \Sigma$
  - $D \in \{L, R\}$,

  s.t. for every $s \in S$, $a \in \Sigma$ there is **at most one** a', D, s' s.t.
  $s \xrightarrow{a/a',D} s'$ is an instruction.
  The elements of I are called **instructions**.
- A element $\llcorner\lrcorner \in \Sigma$ called the symbol for blank (sometimes written B),
- A element $s_0 \in S$ called the initial state.

Tuple Notation

- A Turing machine consists of the 5 components Σ, S, I, $\llcorner\lrcorner$, s_0.
- So a data type of Turing machines is given by these 5 components.
- It is common to say therefore that a Turing Machine is equal to the **tuple** consists of 5 elements, also called **5-tuple** or **quintuple**

$$(\Sigma, S, I, \llcorner\lrcorner, s_0)$$

- Tuples generalise pairs to having more than 2 components. A pair is a 2-tuple.

Meaning of Instructions

- An instruction $s \xrightarrow{a/a',D} s'$ means the following:
  - If the Turing machine is in state s, and the symbol at position of the head is a, then
    - the symbol at this position is changed to a',
    - if $D = \mathrm{L}$, the head moves left,
    - if $D = \mathrm{R}$, the head moves right,
    - and the state is changed to s'.

Example:

$s_0 \xrightarrow{1/0,\mathrm{R}} s_1$

$s_1 \xrightarrow{6/7,\mathrm{L}} s_2$

···	1	5	.	1	6	=				CR								1	5		CR	···	···	1
				↑																				
				s_0																				

Meaning of Instructions

- Note that for the above it is important that for every $s \in S$, $a \in \Sigma$ there is **at most one** a', D, s' s.t. $s \xrightarrow{a/a',D} s'$ is an instruction.
  - Without this condition, there might be more than one choice of selecting a new tape symbol, next state and direction.
  - If we omit this condition, we obtain a **non-deterministic TM**. In this case the machine selects in each step one of the possible choices (provided there exist one) at random.
- If the Turing machine is in a state s and reads symbol a at his head, and there are no s', a', D s.t. $s \xrightarrow{a/a',D} s'$ is an instruction, then the Turing machine stops.

Remark

Until 2011/12, the instructions were denoted as

$$(s, a, s', a', D)$$

instead of

$$s \xrightarrow{a/a',D} s'$$

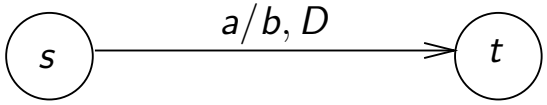
Please take this into account when looking at past exam papers.

TM Architecture vs. TM Program

- As for URMs a TM means both the TM architecture and the TM program.
  - The TM architecture describes that a TM has a tape, a head, a state, and how it is executed.
  - The TM program consists of the alphabet on the tape, the set of states, the instructions, the symbol for blank and the initial state.
- When asked to define a TM which has a certain behaviour one usually actually asks for a TM program, such that a TM with this program has this behaviour.

Visualisation of TMs

- A TM
$$(\Sigma, S, I, \llcorner\lrcorner, s_0)$$
can be visualised by a labelled graph as follows:
  - Vertices: states (i.e. S).
  - Edges: If $s \xrightarrow{a/b,D} t$ is an instruction, then there is an edge



- Furthermore we write an arrow to the initial state coming from nowhere.
- If there are several vertices from s to s', one draws only one arrow with one label for each vertex.

Example

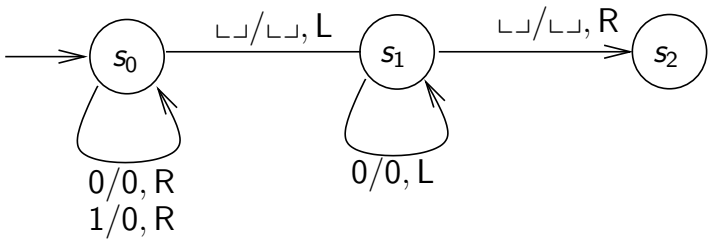
The Turing machine with initial state s_0 and instructions

$s_0 \xrightarrow{0/0,R} s_0$ $\quad s_0 \xrightarrow{1/0,R} s_0$

$s_0 \xrightarrow{\llcorner\lrcorner/\llcorner\lrcorner,L} s_1$ $\quad s_1 \xrightarrow{0/0,L} s_1$

$s_1 \xrightarrow{\llcorner\lrcorner/\llcorner\lrcorner,R} s_2$

is visualised as follows:



Example

- The TM on the previous slide sets the binary number the head is pointing to to zero, provided to the left of the head there are is a blank.
- **Exercise**:
  - This example assumes that the TM points to the left most digit of a binary number.
  - Modify this TM, so that it works as well if the TM points initially to any digit of a binary number.

Equivalent Representations

- The pictorial representation is equivalent to the set of instructions plus an initial state.
- Therefore a TM can both be given by listing its instructions and by the pictorial representation.
- Furthermore the only relevant sets of instructions are those occurring in the pictorial representation. Similarly for the set of symbols on the tape.
- Therefore, assuming that the blank symbol is canonical, we can take the pictorial representation as the complete definition of a TM (with states being the set of states occurring in the diagram, and alphabet consisting of the canonical blank symbol and the states occurring in the diagram).

Notation: bin

- TMs usually operate on binary numbers.
- Therefore we define for a natural number bin(n) as the string in $\{0,1\}^*$ representing the unique normalised binary representation of n.
- **Normalised** means that the string has no leading zeros, except for the string "0" representing 0.
- Furthermore the empty string is not normalised (but is considered as a non-normalised representation of 0).
- Examples:
  - bin(0) = "0",
  - bin(1) = "1",
  - bin(2) = "10",
  - bin(3) = "11",
  - bin(4) = "100", etc.

Notation $(b_0, \ldots, b_{k-1})_2$

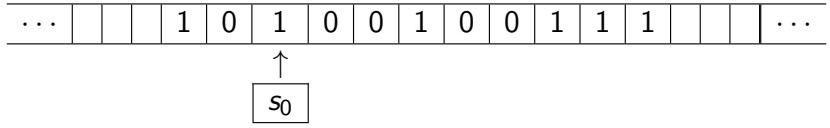
- When interpreting the content of a tape, we need to interpret arbitrary strings of 0 and 1 as natural numbers.
- We usually won't have that this string is normalised (i.e. does not have leading zeros).
- We define
  - $(b_0, \ldots, b_{k-1})_2$ for the natural number having binary representation $b_0, \ldots, b_{k-1}$), e.g.
    $$(01010)_2 = 10$$
  - We allow for leading zeros, so
    $$(001010)_2 = (01010)_2 = (1010)_2 = 10$$
  - We allow as well the empty string
    $$()_2 = 0$$

Example of a TM

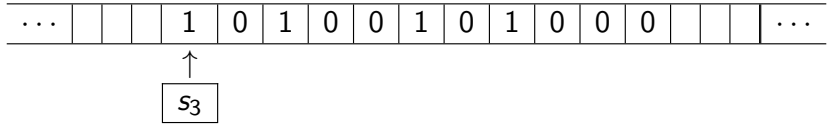
- Development of a TM with $\Sigma = \{0, 1, ␣\}$,
  - where ␣ is the symbol for the blank entry.
- Functionality of the TM:
  - Assume initially the following:
    - The tape contains binary number,
    - The rest of the tape contains ␣.
    - The head points to any digit of the number.
    - The TM in state s_0.
  - Then the TM stops after finitely many steps and then
    - the tape contains the original number incremented by one,
    - the rest of tape contains ␣,
    - the head points to most significant bit.

Example

Initially



Finally



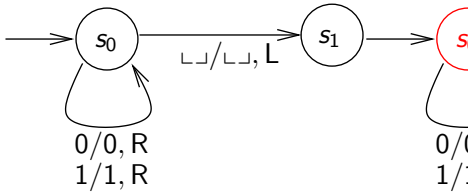
Construction of the TM

- TM is $(\{0, 1, \llcorner\lrcorner\}, S, I, \llcorner\lrcorner, s_0)$.
- States S and instructions I developed in the following.

Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is $\llcorner\lrcorner$, move head left, leave symbol again as it is.
- Achieved by the following instructions:

$$
\begin{array}{lll}
s_0 \xrightarrow{0/0,R} s_0 & & \\
s_0 \xrightarrow{1/1,R} s_0 & & \\
s_0 \xrightarrow{\llcorner\lrcorner/\llcorner\lrcorner,L} s_1 s_0 & s_0 \xrightarrow{0/0,R} s_0 & \\
s_0 \xrightarrow{1/1,R} s_0 & & \\
s_0 \xrightarrow{\llcorner\lrcorner/\llcorner\lrcorner,L} s_1 s_0 & s_0 \xrightarrow{0/0,R} s_0 & \\
s_0 \xrightarrow{1/1,R} s_0 & & \\
s_0 \xrightarrow{\llcorner\lrcorner/\llcorner\lrcorner,L} s_1 & &
\end{array}
$$



Step 2

Increasing a binary number b done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (\underbrace{111\cdots111}_{k \text{ times}})_2$.
  - $b+1 = (1\underbrace{000\cdots000}_{k \text{ times}})_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That's what happens when we add by hand:

$$
\begin{array}{cccccccc}
 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
+ & & & & & & & 1 \\
\hline
 & & & 1 & 1 & 1 & 1 & \\
\hline
 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
$$

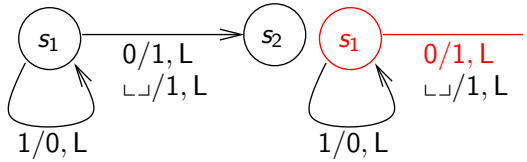
Step 2

- **Otherwise**:
  - Then the representation of the number contains at the end one 0 followed by ones only.
    Includes case where the least significant digit is 0.
    - Example 1: $b = (010001\mathbf{0111})_2$, one 0 followed by 3 ones.
    - Example 2: $b = (010001001\mathbf{0})_2$, least significant digit is 0.
  - Let $b = (b_0 b_1 \cdots b_k 0 \underbrace{111\cdots111}_{l \text{ times}})_2$.
  - $b+1$ obtained by replacing the final block of ones by 0 and the 0 by 1: $b+1 = (b_0 b_1 \cdots b_k 1 \underbrace{000\cdots000}_{l \text{ times}})_2$.

Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a ␣, which is replaced by a 1.
- So we need a new state s_2, and the following instructions

$$s_1 \xrightarrow{1/0,L} s_1$$
$$s_1 \xrightarrow{0/1,L} s_2$$
$$s_1 \xrightarrow{␣/1,L} s_2 s_1 \xrightarrow{1/0,L} s_1$$
$$s_1 \xrightarrow{0/1,L} s_2$$
$$s_1 \xrightarrow{␣/1,L} s_2 s_1 \xrightarrow{1/0,L} s_1$$
$$s_1 \xrightarrow{0/1,L} s_2$$
$$s_1 \xrightarrow{␣/1,L} s_2$$

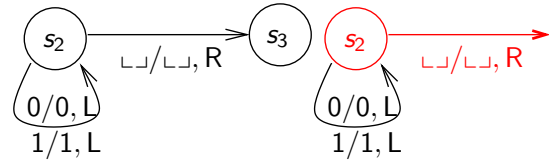


- At the end the head will be one field to the left of the 1 written, and the state will be s_2.

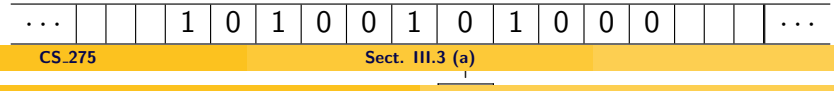
Step 3

Finally, we have to move the most significant bit, which is done as follows

$$s_2 \xrightarrow{0/0,L} s_2$$
$$s_2 \xrightarrow{1/1,L} s_2$$
$$s_2 \xrightarrow{␣/␣,R} s_3 s_2 \xrightarrow{0/0,L} s_2$$
$$s_2 \xrightarrow{1/1,L} s_2$$
$$s_2 \xrightarrow{␣/␣,R} s_3 s_2 \xrightarrow{0/0,L} s_2$$
$$s_2 \xrightarrow{1/1,L} s_2$$
$$s_2 \xrightarrow{␣/␣,R} s_3$$



The program terminates in state s_3.

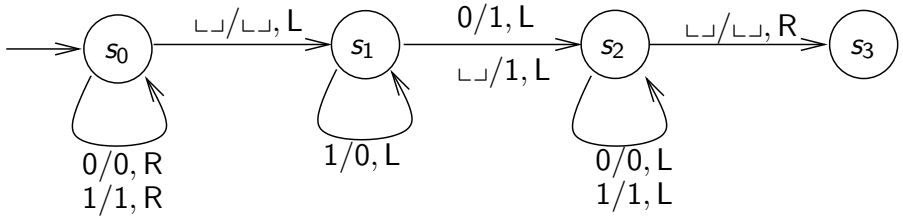


Complete TM

The complete TM is as follows:

$$(\{0, 1, ␣\},$$
$$\{s_0, s_1, s_2, s_3\},$$
$$\{s_0 \xrightarrow{0/0,R} s_0,\ s_0 \xrightarrow{1/1,R} s_0,\ s_0 \xrightarrow{␣/␣,L} s_1,$$
$$s_1 \xrightarrow{1/0,L} s_1,\ s_1 \xrightarrow{0/1,L} s_2,\ s_1 \xrightarrow{␣/1,L} s_2,$$
$$s_2 \xrightarrow{0/0,L} s_2,\ s_2 \xrightarrow{1/1,L} s_2,\ s_2 \xrightarrow{␣/␣,R} s_3\ \},$$
$$␣,$$
$$s_0\ )$$

Complete TM



Function Computed by a TM

Definition (3.1)

Let $\mathrm{T} = (\Sigma, \mathrm{S}, \mathrm{I}, \llcorner\lrcorner, s_0)$ be a Turing machine with $\{0,1\} \subseteq \Sigma$.
Define for every $k \in \mathbb{N}$ $\mathrm{T}^{(k)} : \mathbb{N}^k \overset{\sim}{\to} \mathbb{N}$, where $\mathrm{T}^{(k)}(a_0, \ldots, a_{k-1})$ is computed as follows:

- **Initialisation:**
  - We write on the tape $\mathrm{bin}(a_0)\llcorner\lrcorner\mathrm{bin}(a_1)\llcorner\lrcorner \cdots \llcorner\lrcorner\mathrm{bin}(a_{k-1})$.
    - E.g. if $k = 3$, $a_0 = 0$, $a_1 = 3$, $a_2 = 2$ then we write $0\llcorner\lrcorner 11\llcorner\lrcorner 10$.
  - All other cells contain $\llcorner\lrcorner$.
  - The head is at the left most bit of the arguments written on the tape.
  - The state is set to s_0.
- **Iteration:** Run the TM, until it stops.

Function Computed by a TM

Definition (Cont) (3.1)

- **Output:**
  - **Case 1:** *The TM stops.*
    Only finitely many cells are non-blank.
    Let tape, starting from the head-position, contain $b_0 b_1 \cdots b_{k-1} c$ *where* $b_i \in \{0,1\}$ *and* $c \notin \{0,1\}$.
    (k *might be* 0*).*
    Let
    $$a = (b_0 b_1 \cdots b_{k-1})_2$$
    Then
    $$\mathrm{T}^{(k)}(a_0, \ldots, a_{k-1}) \simeq a \ .$$

Function Computed by a TM

Example: Let $\Sigma = \{0, 1, a, b, \llcorner\lrcorner\}$ where $0, 1, a, b, \llcorner\lrcorner$ are different.

- If the tape starting with the head is as follows:
  - $01001\llcorner\lrcorner 0101\llcorner\lrcorner$
  - or $01001a\llcorner\lrcorner$,

  output is $(01001)_2 = 9$.
- If tape starting with the head is as follows:
  - $ab\llcorner\lrcorner$
  - or a,
  - or $\llcorner\lrcorner$,

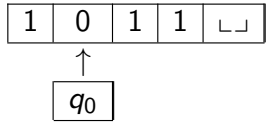
  the output is $()_2 = 0$.

Definition (Cont) (3.1)

- **Case 2:** *Otherwise.*
  Then $\mathrm{T}^{(k)}(a_0, \ldots, a_{k-1})\uparrow$, *i.e.* $\mathrm{T}^{(k)}(a_0, \ldots, a_{k-1}) \simeq \perp$.

Remark

- If the TM terminates with the head in the middle of a binary number, only the portion of this number starting with the head counts.
- Example: Assume the TM terminates with the following configuration:



Then the output is $(011)_2$ which is 3.

Definition Turing Computable Function

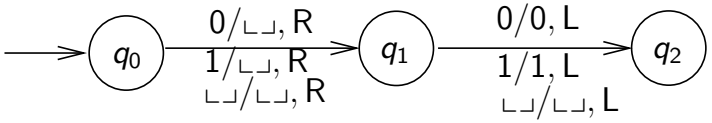
Definition (3.2)

$f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ is **Turing-computable**, in short **TM-computable**, if $f = \mathrm{T}^{(k)}$ for some TM T, the alphabet of which contains $\{0, 1\}$.

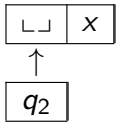
Example: That succ : $\mathbb{N} \xrightarrow{\sim} \mathbb{N}$ and zero : $\mathbb{N} \xrightarrow{\sim} \mathbb{N}$ are Turing-computable was shown above.

Simpler Solution for zero

- zero can be defined in a simpler way by defining a TM which writes a blank and moves right, then moves back (left) and stops with the head pointing to this blank:



The final state of this TM, run with input some binary number, is as follows (x is 0, 1 or $\llcorner\lrcorner$):



Simpler Solution for zero

- The output of $\mathrm{T}^{(1)}(x)$ is the value of largest binary string in the final configuration starting with the head position.
- This string is the empty string, which is interpreted as 0.

Even Simpler Solution

- There are even simpler TMs for defining zero:
  - One which uses only 2 states.
  - and one which uses only 1 state.

Remark

- If the tape of the Turing machine initially contains only finitely many cells which are not blank, then at any step during the execution of the TM only finitely many cells are non blank.
  - Follows since in each step at most one cell can be modified to become non-blank.
  - So in finitely many steps only finitely many cells can be converted from blank to non-blank.

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

(b) Equivalence of URM computable and Turing computable functions

Theorem (3.3)

$f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$ is URM-computable iff it is Turing-computable by a TM with alphabet $\{0, 1, \llcorner\lrcorner\}$.

Proof Idea URM-Computable $\Rightarrow$ TM-Computable

The idea that URM computable functions are TM computable is as follows:

- A URM changes only finitely many registers.
- Therefore it suffices to simulate a URM with only finitely many registers $\mathrm{R}_0, \ldots, \mathrm{R}_n$
- If $\mathrm{R}_0, \ldots, \mathrm{R}_n$ contain values $x_0, \ldots, x_n$, then this state of the URM can be represented by having

$$\mathrm{bin}(x_0)\llcorner\lrcorner\mathrm{bin}(x_1)\llcorner\lrcorner\cdots\llcorner\lrcorner\mathrm{bin}(x_n)$$

  on the tape (surrounded by blanks) and the head pointing to the left most digit of $\mathrm{bin}(x_0)$.
- We can now write TM instructions which take this configuration and executes one URM instruction.

Proof Idea URM-Computable ⇒ TM-Computable

- Instruction $\mathrm{R}_k := \mathrm{R}_k + 1$ can be simulated by
  - moving the head to the kth number
  - incrementing it by 1
  - moving the head back to the left most digit of the first number,
  - and continuing with the simulation of the next instruction following this instruction
    (or terminating if there is no such instruction).
- It might happen that the number of digits of the number incremented increases.
  - In this case first shift the numbers to the left once to the left.

Proof Idea URM-Computable ⇒ TM-Computable

- Instruction $\mathrm{R}_k := \mathrm{R}_k \dot{-} 1$ can be simulated similarly.
- Instruction `if` $\mathrm{R}_k = 0$ `then goto` l can be simulated by checking whether the kth number is zero or not.
  - If it is zero continue executing the simulation of instruction l.
  - If it is not zero continue executing the next instruction.
  - If in one of these cases the instruction doesn't exist, terminate.

Proof Idea URM-Computable ⇒ TM-Computable

- Let the original URM be U and the resulting TM be T.
- $\mathrm{T}^{(k)}$ will write the arguments in binary on the tape.
  - The arguments will just be written in the register positions.
- Then T will simulate U.
- T will terminate iff U terminates.
- If T terminates $\mathrm{T}(k)(x_0, \ldots, x_{k-1})$ will return the binary value of the first number of the tape which is the content of R_0 and therefore the output of the $\mathrm{U}^{(k)}(x_0, \ldots, x_{k-1})$.
- So $\mathrm{T}^{(k)}$ and $\mathrm{U}^{(k)}$ return the same results.
- Details can be found in the proof of Lemma 3.4 below.

Proof Idea TM-Computable ⇒ URM-Computable

- At any time during the execution of a TM only a finite portion of the tape is non-blank.
- Therefore the state of a TM can be encoded by giving
  - the finite portion of the tape which is non-blank,
  - the position of the head in this portion,
  - the state of the TM
- There are techniques for encoding this in a computable way as a natural number.
- Now simulate the TM by a URM in a similar way as the simulation of a URM by a TM.

Formal Proof

A formal proof of one direction (URM-computable functions are Turing computable) can be found in the additional material (Lemma 3.4).

Notation $\vec{x}, \vec{y}$ etc.

- In many expressions we will have arguments, to which we don't refer explicitly.
  Example: Variables $x_0, \ldots, x_{n-1}$ in

$$f(x_0, \ldots, x_{n-1}, y) = \begin{cases} g(x_0, \ldots, x_{n-1}), & \text{if } y = 0, \\ h(x_0, \ldots, x_{n-1}), & \text{if } y > 0. \end{cases}$$

- We abbreviate $x_0, \ldots, x_{n-1}$, by $\vec{x}$.
- Then the above can be written shorter as

$$f(\vec{x}, y) = \begin{cases} g(\vec{x}), & \text{if } y = 0, \\ h(\vec{x}), & \text{if } y > 0. \end{cases}$$

- In general, $\vec{x}$ stands for $x_0, \ldots, x_{n-1}$, where the number of arguments n is clear from the context.

Examples

- If

$$f : \mathbb{N}^{n+1} \to \mathbb{N}$$

  then in $f(\vec{x}, y)$,
  $\vec{x}$ needs to stand for n arguments.
  Therefore

$$\vec{x} = x_0, \ldots, x_{n-1}$$

- If

$$f : \mathbb{N}^{n+2} \to \mathbb{N}$$

  then in $f(\vec{x}, y)$,
  $\vec{x}$ needs to stand for $n + 1$ arguments,
  so

$$\vec{x} = x_0, \ldots, x_n$$

Examples

- If P is an $n + 4$-ary relation, then in $P(\vec{x}, y, z)$,
  $\vec{x}$ stands for

$$x_0, \ldots, x_{n+1}$$

- Similarly, we write $\vec{y}$ for

$$y_0, \ldots, y_{n-1}$$

  where n is clear from the context.
- Similarly for

$$\vec{z}, \vec{n}, \vec{m}, \ldots$$

Notation

- $$\forall \vec{x} \in \mathbb{N}.\varphi(\vec{x})$$

  stands for

  $$\forall x_0, \ldots, x_{n-1} \in \mathbb{N}.\varphi(x_0, \ldots, x_{n-1})$$

  where the number of variables n is implicit (and usually unimportant).
- $$\exists \vec{x} \in \mathbb{N}.\varphi(\vec{x})$$

  is to be understood similarly.

Extension to Arbitrary Alphabets

- Let A be a finite alphabet s.t. $\llcorner\lrcorner \notin A$, and $B := A^*$.
- To a Turing machine $T = (\Sigma, S, I, \llcorner\lrcorner, s_0)$ with $A \subseteq \Sigma$ corresponds a partial function $T^{(A,n)} : B^n \xrightarrow{\sim} B$, where $T^{(A,n)}(a_0, \ldots, a_{n-1})$ is computed as follows:
  - Initially write $a_0 \llcorner\lrcorner \cdots \llcorner\lrcorner a_{n-1}$ on the tape, otherwise $\llcorner\lrcorner$.
    Start in state s_0 on the left most position of a_0.
  - Iterate TM as before.
  - In case of termination, the output of the function is $c_0 \cdots c_{l-1}$, if the tape contains, starting with the head position $c_0 \cdots c_{l-1} d$ with $c_i \in A$, $d \notin A$.
  - Otherwise, the function value is undefined.

Extension to Arbitrary Alphabets

- Notion is modulo encoding of A^* into $\mathbb{N}$ equivalent to the notion of Turing-computability on $\mathbb{N}$.
- However, when considering complexity bounds, this notation might be more appropriate.
  - Avoids encoding/decoding into $\mathbb{N}$.

Characteristic function

- We define for an n-ary relation M (i.e. a subset of $\mathbb{N}^n$) a function

  $$\chi_M : \mathbb{N}^n \to \mathbb{N}$$

  which decides for $\vec{x} \in \mathbb{N}^n$ whether $M(\vec{x})$ holds
  (Here χ is a Greek letter pronounced "chi".
  $\vec{x}$ stands for arguments $x_1, \ldots, x_n$).
- Formally the **characteristic function for M** χ_M is defined as follows:

  $$\chi_M(\vec{x}) := \begin{cases} 1 & \text{if } M(\vec{x}) \text{ holds,} \\ 0 & \text{otherwise} \end{cases}$$
- If we treat true as 1 and false as 0, then the characteristic function decides whether $M(\vec{x})$ holds or not:

  $$\chi_M(\vec{x}) = \begin{cases} \text{true} & \text{if } M(\vec{x}) \text{ holds,} \\ \text{false} & \text{otherwise} \end{cases}$$

Turing-Computable Predicates

- A predicate A is Turing-decidable, iff χ_A is Turing-computable.
- Instead of simulating χ_A
  - means to write the output of χ_A (a binary number 0 or 1) on the tape

  it is more convenient, to take TM with two additional special states s_{true} and s_{false} corresponding to truth and falsity of the predicate.

Turing-Computable Predicates

- Then a predicate is Turing decidable, if, when we write initially the inputs as before on the tape and start executing the TM,
  - it always terminates in s_{true} or s_{false},
  - and it terminates in s_{true}, iff the predicate holds for the inputs,
  - and in s_{false}, otherwise.
- The latter notion is equivalent to the first notion.
- Usually the latter one is taken as basis for complexity considerations.

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

(c) Undecidability of the Turing Halting Problem

- Undecidability of the Halting Problem first proved 1936 by Alan Turing.
- In this Section, we will identify computable with Turing-computable.
  - This will later be justified by the Church-Turing thesis.

History of Computability Theory

Alan Mathison Turing
(1912 – 1954)
Introduced the Turing machine.
Proved the undecidability
of the Turing-Halting problem.

Definition of Problem

Definition (3.5)

(a) A **problem** is an n-ary predicate $M(\vec{x})$ of natural numbers, i.e. a property of n-tuples of natural numbers.

(b) A problem (or predicate) M is (Turing-)**decidable**, if **the characteristic function χ_M of M** is (Turing-)computable.

(The characteristic function χ_M was defined at the End of Subsect. 3 (b)).

Example of Decidable Problems

- The binary predicate

$$\text{Multiple}(x, y) :\Leftrightarrow x \text{ is a multiple of } y$$

  is a predicate and therefore a problem.
- $\chi_{\text{Multiple}}(x, y)$ decides, whether $\text{Multiple}(x, y)$ holds (then it returns 1 for yes), or not:

$$\chi_{\text{Multiple}}(x, y) = \begin{cases} 1 & \text{if } x \text{ is a multiple of } y, \\ 0 & \text{if } x \text{ is not a multiple of } y. \end{cases}$$

- χ_{Multiple} is intuitively computable, therefore Multiple is decidable.

Need of Encoding of TMs

- We want to show that it is not decidable whether a Turing Machine terminates or not.
- For this we need to be able to talk about programs which have as input a Turing Machine.
- For this we need to give a formalisation of what a Turing Machine is.
- Since we are restricting ourselves to functions having as arguments elements of $\mathbb{N}^k$, we need to encode a TM as an element of $\mathbb{N}^k$ for some k.
- We will actually encode TMs as elements of $\mathbb{N}$.

Encoding of Turing Machines

- A Turing Machine is a quintuple (or five-tuple) $(\Sigma, S, I, \llcorner\lrcorner, s_0)$.
- We can assume that $\llcorner\lrcorner$, each symbol of the alphabet, and each state can be represented by a string of letters and numbers.
- Then this quintuple can be written as a string of ASCII-symbols.
- $\Rightarrow$ Turing machines can be represented as elements of A^*, where $A =$ set of ASCII-symbols.
- There are computable functions, which allow to encode strings as natural numbers and corresponding computable decoding functions.
  - Taught in an extended module on computability theory.
- $\Rightarrow$ Turing machines can be encoded as natural numbers.
- When carrying out details of the above one usually refers to more sophisticated encodings.

Encoding of Turing Machines

- Let for a Turing machine T, $\mathrm{encode}(\mathrm{T}) \in \mathbb{N}$ be its code.
- It is intuitively decidable, whether a string of ASCII symbols is a Turing machine.
  - One can show that this can be decided by a Turing machine.
- $\Rightarrow$ It is intuitively decidable, whether $n = \mathrm{encode}(\mathrm{T})$ for a Turing machine T.

$\{e\}^k(n)$

- Assume $e \in \mathbb{N}$. We define a partial function $\{e\}^k : \mathbb{N}^k \stackrel{\sim}{\to} \mathbb{N}$, by

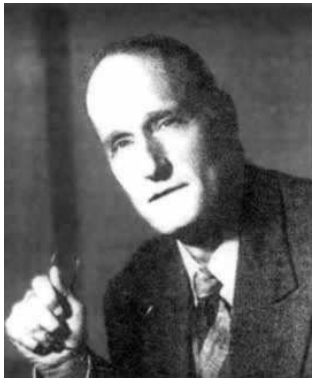
$$\{e\}^k(\vec{x}) \simeq \begin{cases} m & \text{if } e = \mathrm{encode}(\mathrm{T}) \text{ for some Turing machine } \mathrm{T} \\ & \text{and } \mathrm{T}^{(k)}(\vec{x}) \simeq m, \\ \perp & \text{otherwise.} \end{cases}$$

- So if $e = \mathrm{encode}(\mathrm{T})$, $\{e\}^k = \mathrm{T}^{(k)}$.
  - Roughly speaking, $\{e\}^k$ is the function computed by the eth Turing machine.
  - So for every computable (more precisely Turing-computable) function $f : \mathbb{N}^k \stackrel{\sim}{\to} \mathbb{N}$ there exists an e s.t. $f = \{e\}^k$.

$\{e\}^k$

- The notation $\{e\}^k$ is due to Stephen Kleene.
- $\{\}$ are called **Kleene-Brackets**.
- We write $\{e\}$ for $\{e\}^1$.

Stephen Cole Kleene



Stephen Cole Kleene
(1909 – 1994)
Probably the most influential computability theorist up to now. Introduced the partial recursive functions.

Definition of the Turing Halting Problem

Definition (3.6)

The **Turing Halting Problem** is the following binary predicate:

$$\mathrm{Halt}(e, n) :\Leftrightarrow \{e\}(n)\downarrow$$

We will show that Halt is undecidable.

Example

- Let $e = \mathrm{encode}(\mathrm{T})$, where T is the Turing machine T which translates the URM program consisting of only one instruction

$$0 : \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 0$$

- If this TM is run with arguments written on the tape, it loops if the first argument is 0, and terminates otherwise with its first argument unchanged.
- So we have

$$\{e\}(k) \simeq \mathrm{T}^{(1)}(k) \simeq \begin{cases} k & \text{if } k > 0 \\ \bot & \text{otherwise.} \end{cases}$$

- Therefore $\mathrm{Halt}(e, k)$ holds for $k > 0$ and does not hold for $k = 0$.

Question

If we fix $e = \mathrm{encode}(\mathrm{T})$ for the Turing machine above, can we decide, for which k we have that $\mathrm{Halt}(e, k)$ holds?

Remark

- Below we will see: Halt is undecidable.
- However, the following function WeakHalt is computable:

$$\text{WeakHalt}(e,n) :\simeq \begin{cases} 1 & \text{if } \{e\}(n)\downarrow \\ \bot & \text{otherwise} \end{cases}$$

  - Computed as follows:
    First check whether $e = \text{encode}(\mathrm{T})$ for some Turing machine T.
    If not, enter an infinite loop.
    Otherwise, simulate T with input n.
    If simulation stops, output 1, otherwise the program loops for ever.

Question

- What is WeakHalt(e,n), where e is a code for the Turing machine, which operates as follows:
  - if the input is 0 it goes into an infinite loop, i.e. never terminates;
  - if the input is not 0 it stops immediately.
- That TM can be obtained by translating the following URM program into a TM:

$$0 : \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 0$$

Undecidability of the Turing Halting Problem

Theorem (3.7)

The Turing halting problem is not Turing-decidable.

Proof:

- **Assume** the Turing Halting problem were Turing-decidable i.e. assume that we can decide using a Turing machine whether $\{e\}(n)\downarrow$ holds.
- We will define below a computable function $f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$, s.t. for all $e \in \mathbb{N}$ we have $f \neq \{e\}$.
- Therefore f cannot be computed by the Turing machine with code e for any e, i.e. f is non-computable.
- Therefore we obtain a **contradiction.**

Diagonalisation Argument

- The values of $\{e\}(n)$ depend on the concrete encoding of Turing Machines used.
- Assume the values of $\{e\}(n)$ are as in the following table:

e \ n	0	1	2	3
0	0	$\bot$	1	5
1	2	$\bot$	0	$\bot$
2	3	0	1	2
3	2	1	$\bot$	$\bot$

- The values shaded in blue form the diagonal.
- We will define $f(e)$ such that it is different from the values $\{e\}(e)$ on the diagonal.

Diagonalisation Argument

-

e \ n	0	1	2	3
0	0	$\perp$	1	5
1	2	$\perp$	0	$\perp$
2	3	0	1	2
3	2	1	$\perp$	$\perp$

- If $\{e\}(e)$ is defined (i.e. $\{e\}(e)\downarrow$) then we make $f(e)$ undefined: $f(e) \simeq \perp$.
- If $\{e\}(e)$ is undefined (i.e. $\{e\}(e)\uparrow$) then we make $f(e)$ defined: $f(e) \simeq 0$.

Diagonalisation Argument

-

e \ n	0	1	2	3
0	0	$\perp$	1	5
1	2	$\perp$	0	$\perp$
2	3	0	1	2
3	2	1	$\perp$	$\perp$

$f(e)$
$\perp$
0
$\perp$
0

- With our values of $\{e\}(n)$ we obtain:
  - $\{0\}(0) \simeq 0$, therefore $\{0\}(0)\downarrow$, therefore we set $f(0) \simeq \perp$.
  - $\{1\}(1) \simeq \perp$, therefore $\{1\}(1)\uparrow$, therefore we set $f(1) \simeq 0$.
  - $\{2\}(2) \simeq 1$, therefore $\{2\}(2)\downarrow$, therefore we set $f(2) \simeq \perp$.
  - $\{3\}(3) \simeq \perp$, therefore $\{3\}(3)\uparrow$, therefore we set $f(3) \simeq 0$.

Diagonalisation Argument

-

e \ n	0	1	2	3
0	0	$\perp$	1	5
1	2	$\perp$	0	$\perp$
2	3	0	1	2
3	2	1	$\perp$	$\perp$

$f(e)$
$\perp$
0
$\perp$
0

- $f(e)$ can be computed assuming that $\text{Halt}(e, n)$ is decidable:
  - If $\text{Halt}(e, e)$ then $\{e\}(e)\downarrow$, $f(e) \simeq \perp$.
  - If $\neg\text{Halt}(e, e)$ then $\{e\}(e)\uparrow$, $f(e) \simeq 0$.
  - So $f(e) \simeq \begin{cases} \perp, & \text{if } \text{Halt}(e, e) \\ 0, & \text{if } \neg\text{Halt}(e, e) \end{cases}$

Diagonalisation Argument

-

e \ n	0	1	2	3
0	0	$\perp$	1	5
1	2	$\perp$	0	$\perp$
2	3	0	1	2
3	2	1	$\perp$	$\perp$

$f(e)$
$\perp$
0
$\perp$
0

- f can be computed.
- Furthermore, we have $f(e) \not\simeq \{e\}(e)$.
- Therefore we have achieved $f \neq \{e\}$ for all e.
- However since f is computable, there must exist an e s.t. $f = \{e\}$.
- We obtain a contradiction.

Proof of Theorem 3.7

- The precise argument is as follows:
- We define $f(e)$ in such a way that $f \neq \{e\}$ by ensuring $f(e) \not\simeq \{e\}(e)$.
- If $\{e\}(e)\downarrow$, then we let $f(e)\uparrow$.
- If $\{e\}(e)\uparrow$, we let $f(e)\downarrow$, e.g. by defining $f(e) \simeq 0$ (any other defined result would be appropriate as well).
- So we define

$$f(e) \simeq \begin{cases} \bot, & \text{if } \{e\}(e)\downarrow \\ 0, & \text{if } \{e\}(e)\uparrow \end{cases} \simeq \begin{cases} \bot, & \text{if } \mathrm{Halt}(e,e) \\ 0, & \text{if } \neg\mathrm{Halt}(e,e) \end{cases}$$

Proof of Theorem 3.7

$$f(e) \simeq \begin{cases} \bot, & \text{if } \{e\}(e)\downarrow \\ 0, & \text{if } \{e\}(e)\uparrow \end{cases} \simeq \begin{cases} \bot, & \text{if } \mathrm{Halt}(e,e) \\ 0, & \text{if } \neg\mathrm{Halt}(e,e) \end{cases}$$

- Since we assumed Halt to be decidable, f is computable (Exercise: show that f is computable by a Turing machine, assuming a Turing machine for Halt).
- Furthermore $f(e)\downarrow \Leftrightarrow \{e\}(e)\uparrow$, therefore $f \neq \{e\}$.
- But then f is not computable, since if it were computable it would be computable by a TM with code e, so would have $f = \{e\}$ for some e.
- So we obtain a contradiction, and obtain therefore that the assumption that Halt is decidable was false.

Proof of Theorem 3.7

The complete proof on one slide is as follows:

- Assume Halt were decidable.
- Define

$$f(e) \simeq \begin{cases} \bot, & \text{if } \{e\}(e)\downarrow \\ 0, & \text{if } \{e\}(e)\uparrow \end{cases}$$

- By Halt decidable, we obtain f is computable, so $f = \{e\}$ for some e.
- But then

$$f(e)\downarrow \overset{\text{Def. of } f}{\Leftrightarrow} \{e\}(e)\uparrow \overset{f=\{e\}}{\Leftrightarrow} f(e)\uparrow$$

Remark

- The above proof can easily be adapted to any reasonable programming language, in which one can define all intuitively computable functions.
- Such programming languages are called **Turing-complete** languages.
  - Babbage's machine was, if one removes the restriction to finite memory, Turing-complete, since it had a conditional jump.
- For standard Turing complete languages, the unsolvability of the Turing-halting problem means:
  it is not possible to write a program, which checks, whether a program on given input terminates.

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

(d) The Church-Turing Thesis

We have introduced two models of computations:

- The URM-computable functions.
- The Turing-computable functions.

Further we have indicated why the two models of computation compute the same partial functions.

The Church-Turing Thesis

Lots of other models of computation have been studied:

- The partial recursive functions.
- The while programs.
- Symbol manipulation systems by Post and by Markov.
- Equational calculi by Kleene and by Gödel.
- The λ-definable functions.
- Any of the programming languages Pascal, C, C++, Java, Prolog, Haskell, ML (and many more).
- Lots of other models of computation.

The Church-Turing Thesis

- One can show that the partial functions computable in these models of computation are again exactly the Turing computable functions.
- So all these attempts to define a complete model of computation result in the same set of partial recursive functions.
- Therefore we arrive at the Church-Turing Thesis, also called Church's thesis.

The Church-Turing Thesis

Church-Turing Thesis:
The (in an intuitive sense) computable partial functions are exactly the Turing-computable functions

(or equivalently the URM-computable functions or equivalently the functions computable in any other known Turing-complete model of computation).

Philosophical Thesis

- This thesis is **not a mathematical theorem**.
- It is a **philosophical thesis**.
- Therefore the Church-Turing thesis **cannot be proven**.
- We can only provide **philosophical evidence** for it.
- This evidence comes from the following **considerations and empirical facts**:

Empirical Facts

- All complete models of computation suggested by researchers define the same set of partial functions.
- Many of these models were carefully designed in order to capture intuitive notions of computability:
  - The Turing machine model captures the intuitive notion of **computation on a piece of paper** in a general sense.
  - The URM machine model captures the general notion of **computability by a computer**.
  - Symbolic manipulation systems capture the general notion of computability by **manipulation of symbolic strings**.

Empirical Facts

- No intuitively computable partial function, which is not partial recursive, has been found, despite lots of researchers trying it.
- A strong intuition has been developed that in principal programs in any programming language can be simulated by Turing machines and URMs.

Because of this, only few researchers doubt the correctness of the Church-Turing thesis.

Decidable Sets

- A predicate A is URM-/Turing-decidable iff χ_A is URM-/Turing-computable.
- A predicate A is decidable iff χ_A is computable.
- By the Church-Turing thesis to be computable is the same as to be URM-computable or to be Turing-computable.
- So the decidable predicates are exactly the URM-decidable and exactly the Turing-decidable predicates.

Halting Problem

- Because of the equivalence of the models of computation, the halting problem for any of the above mentioned models of computation is undecidable.
- Especially it is undecidable, whether a program in one of the programming languages mentioned terminates:
  - Assume we had a decision procedure for deciding whether or not say a Java program terminates for given input.
  - Then we could, using a translation of URMs into Java programs, decide the halting problem for URMs, which is impossible.

III.3 (a) Definition of the Turing Machine

III.3 (b) Equivalence of URM and Turing computable functions

III.3 (c) Undecidability of the Turing Halting Problem

III.3 (d) The Church-Turing Thesis

III.3 (e) Total Programming Languages, Interactive Programs

Programming Languages with Total Functions

Theorem

Assume a programming language such that

- *all functions definable are total.*
- *all programs can be encoded as natural numbers,*
- *we can decide for program with code e whether it defines a function* $\mathbb{N} \to \mathbb{N}$,
- *if program with code e defines a function* $\mathbb{N} \to \mathbb{N}$, *we can compute compute from e and k the result of applying this function to k.*

Then there exists a total computable function, which is not computable in this language.

Proof

- By program e we mean in the following program with code e.
- Define

$$h : \mathbb{N}^2 \to \mathbb{N}$$
$$h(e,k) = \begin{cases} 0 & \text{if program } e \text{ is not a unary function} \\ l & \text{if program } e \text{ is a unary function} \\ & \text{which applied to } k \text{ returns } l \end{cases}$$

- h is computable.
- Define

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(e) = h(e,e) + 1$$

- f is computable.

Proof

- There is no program e of the language which computes f:
  - Assume e computed f.
  - Then we had
    - $h(e,e) = f(e)$ (by e computes f)
    - $f(e) = h(e,e) + 1$ (by definition of f),
    - therefore $h(e,e) = f(e) = h(e,e) + 1$, a contradiction.
  - Therefore no program e computes f.

Programs which operate on other data types than $\mathbb{N}$

- Above we only considered computable functions $\mathbb{N}^n \to \mathbb{N}$.
- One might think of dealing with computable functions $f : A \to B$ where A and B are arbitrary data types.
- First of all we need to restrict ourselves to data types A, B which can be represented on the computer.
- Every element of such data type needs to be represented in computer memory as a sequence of binary words.
- All these binary words together form one long binary word, which encodes a binary number.
  - To avoid problem with leading zeros, one might add a 1 to the beginning of the word.
- These binary numbers are natural numbers.
- Therefore the function $f : A \to B$ can be replaced by a function $f : \mathbb{N} \to \mathbb{N}$.

Programs which operate on other data types than $\mathbb{N}$

- The above doesn't show how to represent data types as natural numbers.
- There is a rich theory on data types which can be encoded in such a way.
  - With different encodings one might get different sets of computable functions.

Interactive Programs

- Above we were only considering computable functions.
- They are given by **batch programs**, programs which have only a fixed finite number of inputs and one output.
- Usually programs are interactive.
- The notion of an interactive program can however be reduced to that of a batch program:

Interactive Programs

- An interactive program is a program which has a state (which can be a natural number) $s \in \mathbb{N}$,
- has an initial state $s_0 \in \mathbb{N}$,
- depending on
  - a state s
  - an encoding of the user input as a natural number (keystrokes, mouse clicks, data from sensors, etc)

  computes
  - the next state of the interactive program
  - an encoding of the output of the programs (output could be values of actuators, changes of graphical output etc)

Interactive Programs

- The halting problem is here the question, whether the interactive program, after a user input computes an output and is ready for the next user input.
- This problem is just the halting problem for batch programs.
- There are various articles by Peter Hancock and Anton Setzer on the IO monad, which explore the above notion of an interactive program.