Sect. 3: The URM

(a) Definition of the URM.
(b) Higher level programming concepts for URMs.
(c) URM computable functions.
(d) Configurations of URMs.
(e) The undecidability of the halting problem.
(a) Definition of the URM

A **model of computation** consists of a set of partial functions together with methods, which describe, how to compute those functions.

One aims at models of computation which are **complete**, which means they compute all intuitively computable functions.

A model of computation is **Turing-complete**, if it is complete in that sense.

Since “intuitively computable” is not a mathematical notion, Turing-completeness is not a mathematical notion and cannot be proved mathematically.
Turing Completeness

Sometimes by “Turing complete” it is meant that the model computes all functions computable on a Turing machine – then one obtains a mathematical definition.
Models of Computation

- Aim: an as simple model of computation as possible: constructs used minimised, while still being able to represent all intuitively computable functions.
  - Makes it easier to show for other models of computation, that the first model can be interpreted in it.
  - In mathematics one always aims at giving as simple and short definitions as possible, and to avoid unnecessary additions.

- Models of computation are mainly used for showing that something is non-computable rather than for showing that something is computable in this model.
The URM (the unlimited register machine) is one model of computation.

- Particularly easy.
- It defines a virtual machine, i.e. a description how a computer would execute its program.
- The URM is not intended for actual implementation (although it can easily be implemented).
- It is not intended to be a realistic model of a computer.
- It is intended as a mathematical model, which is then investigated mathematically.
- Not many programs are actually written in it – one shows that in principal there is a way of writing a certain program in this language.
The URM

- Rather difficult to write actual programs for the URM.
- Low level programming language (only goto)
- URM idealised machine – no bounds on the amount of memory or execution time
  - however all values will be finite.
- Many variants of URM – this URM will be particularly easy.
Description of the URM

The URM consists of

- infinitely many registers $R_i$
  - can store arbitrarily big natural number;
- a finite sequence of instructions $I_0, I_1, I_2, \ldots I_n$;
- and a program counter PC:
  - stores a natural number.
  - If PC contains a number $0 \leq i \leq n$, it points to instruction $I_i$.
  - If content of PC is outside this range, the program stops.
John Shepherdson (Bristol) (2nd from the right)
Developed together with Sturgis the URM.
The URM

\[ R_0 \quad R_1 \quad R_2 \quad R_3 \quad R_4 \quad R_5 \quad R_6 \quad R_7 \quad R_8 \quad \cdots \]

\[ I_0 \quad I_1 \quad I_2 \quad \cdots \quad I_n \]

Execute Instruction
The URM

\[ \begin{array}{cccccccccc}
R_0 & R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 & R_8 & \cdots \\
\end{array} \]

\[ \begin{array}{cccccc}
I_0 & I_1 & I_2 & \cdots & I_n \\
\end{array} \]

PC

Program has terminated
3 kinds of **URM instructions**.

- The **successor instruction**

\[ \text{succ}(k) \]

where \( k \in \mathbb{N} \).

- Execution:
  - Add 1 to register \( R_k \).
  - Increment PC by 1.
  - \( \rightarrow \) execute next instruction or terminate.

- A more readable notation is

\[ R_k := R_k + 1 \]
The predecessor instruction

\[ \text{pred}(k) \]

where \( k \in \mathbb{N} \).

Execution:
- If \( R_k \) contains value \( > 0 \), decrease the content by 1.
- If \( R_k \) contains value 0, leave it as it is.
- In all cases increment PC by 1.

A more readable notation is

\[ R_k := R_k - 1 \]
Here

\[ x \rhd y := \max\{x - y, 0\} \]

i.e.

\[
x \rhd y = \begin{cases} 
  x - y & \text{if } y \leq x, \\
  0 & \text{otherwise}.
\end{cases}
\]
URM Instructions

The **conditional jump instruction**

\[
\text{ifzero}(k, q)
\]

where \( k, q \in \mathbb{N} \). Execution:

- If \( R_k \) contains 0, PC is set to \( q \)
  - next instruction is \( I_q \), if \( I_q \) exists.
    - If no instruction \( I_q \) exists, the program stops.
- If \( R_k \) does not contain 0, the PC incremented by 1.
  - Program continues executing the next instruction, or terminates, if there is no next instruction.
- A more readable notation is

\[
\text{if } R_k = 0 \text{ then goto } q
\]
Finiteness

A URM program refers only to *finitely many registers*, namely those referenced explicitly in one of the instructions.
Example of a Urm Program

The following is an example of a URM-program:

\[ I_0 = \text{ifzero}(0, 3) \]
\[ I_1 = \text{pred}(0) \]
\[ I_2 = \text{ifzero}(1, 0) \]

We can write it in more readable form as follows:

0: if R_0 = 0 then goto 3
1: R_0 := R_0 - 1
2: if R_1 = 0 then goto 0
Example of a Urm Program

0: if $R_0 = 0$ then goto 3
1: $R_0 := R_0 \div 1$
2: if $R_1 = 0$ then goto 0

Assume $R_1$ is initially zero.
Then $R_1$ will never be changed by the program, so it will remain 0 for ever.
So in instruction 2 the URM will always jump to instr. 0.
Then the program will as long as $R_0 \neq 0$ decrease $R_0$ by 1.
The result is that $R_0$ is set to 0.
This corresponds to the instruction from a higher level language $R_0 := 0$. 
For every $U$-program we define the function defined by it.

In fact there are many function which are defined by the same $U$-program:

- A unary function $U^{(1)}$, which stores its argument in $R_0$, sets all other registers to 0, then starts to run the $U$.
  - If the $U$ stops, the result is read off from $R_0$.
  - Otherwise the result is undefined.

- A binary function $U^{(2)}$, which stores its two arguments in $R_0$ and $R_1$, then operates as $U^{(1)}$.
  - And so on. In general we obtain a $k$-ary partial function $U^{(k)}$ for every $k \geq 1$. 
Definition \( U^{(k)} \)

- Let \( U = I_0, \ldots, I_n \) be a URM program, \( k \in \mathbb{N}, k \geq 1 \).
- We define a function

\[
U^{(k)} : \mathbb{N}^k \rightarrow \mathbb{N}
\]

by determining how it is computed:

- Assume we want to compute \( U^{(k)}(a_0, \ldots, a_{k-1}) \).

**Initialisation:**

- PC set to 0.
- \( a_0, \ldots, a_{k-1} \) stored in registers \( R_0, \ldots, R_{k-1} \), respectively.
- All other registers set to 0.
  (Sufficient to do this for registers referenced in the program).
Iteration:
As long as the PC points to an instruction, execute it. Continue with the next instruction as given by the PC.

Output:
If PC value > \( n \), the program stops.
- The function returns the value in \( R_0 \).
- So if \( R_0 \) contains \( b \) then

\[
U^{(k)}(a_0, \ldots, a_{k-1}) \sim b.
\]

If the program never stops,

\[
U^{(k)}(a_0, \ldots, a_{k-1}) \uparrow.
\]
URM-Computable Functions

\[ f : \mathbb{N}^k \sim \rightarrow \mathbb{N} \text{ is URM-computable, if } f = U^{(k)} \text{ for some } k \in \mathbb{N} \text{ and some URM program } U. \]
Change of Notation

In previous years, $P$ was used instead of $U$ to denote URM programs.

$P$ will be used for Turing machines.

In order to distinguish URM-programs and Turing machine programs, we write here $U$ instead of $P$.

Please take this into account when looking at exams and slides from previous years.
Partial Computable Functions

For a partial function $f$ to be computable we need only:

- If $f(a) \downarrow$, then after finite amount of time we can determine this property, and the value of $f(a)$.

- If $f(a) \uparrow$, we will wait infinitely long for an answer, so we never determine that $f(a) \uparrow$.

Turing halting problem is the question: “Is $f(a) \downarrow$?”. Turing halting problem is undecidable.

If we want to have always an answer, we need to refer to total computable functions.
Partial Computable Functions

In order to describe the total computable functions, we need to introduce the partial computable functions first.

There is no program language s.t.

- it is decidable whether a string is a program,
- and the program language describes all total computable functions.

- This is essentially a consequence of the undecidability of the Turing Halting Problem.
Example of URM-Comp. Functions

\[ f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}, \quad f(x) \sim 0 \] is URM computable. We derive a URM-program for it in several steps.

**Step 1:**
Initially \( R_0 \) contains \( x \) and the other registers contain 0. Program should then terminate with \( R_0 \) containing 0. A higher level program is as follows:

\[ R_0 := 0 \]
Example of URM-Comp. Functions

\[ f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}, \, f(x) \equiv 0 \]

**Step 2:**
Only successor and predecessor available, replace the program by the following:

\[
\text{while } R_0 \neq 0 \text{ do } \{R_0 := R_0 - 1\}
\]
Example of URM-Comp. Functions

\( f : \mathbb{N} \rightarrow \mathbb{N}, f(x) \sim 0 \)

**Step 3:**
Replace while-loop by a goto:

LabelBegin: \( \text{if } R_0 = 0 \text{ then goto LabelEnd;} \)
\( R_0 := R_0 + 1; \)
\( \text{goto LabelBegin;} \)

LabelEnd:
Example of URM-Comp. Functions

\[ f : \mathbb{N} \rightarrow \mathbb{N}, \ f(x) \approx 0 \]

**Step 4:**
Replace last goto by a conditional goto, depending on \( R_1 = 0 \).
\( R_1 \) is initially true and never modified, therefore this jump will always be carried out.

\[
\text{LabelBegin : if } R_0 = 0 \text{ then goto LabelEnd;}
R_0 := R_0 \div 1;
\text{if } R_1 = 0 \text{ then goto LabelBegin;}
\]

\text{LabelEnd :}
Example of URM-Comp. Functions

\[ f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}, \ f(x) \sim 0 \]

**Step 5:**
Translate the program into a URM program \( I_0, I_1, I_2: \)

\[
\begin{align*}
I_0 &= \text{ifzero}(0, 3) \\
I_1 &= \text{pred}(0) \\
I_2 &= \text{ifzero}(1, 0)
\end{align*}
\]
Remark on Jump Addresses

When inserting URM programs \( U \) as part of new URM programs, jump addresses will be adapted accordingly.

E.g. in

\[
\begin{align*}
\text{succ}(0) \\
\text{U} \\
\text{pred}(0)
\end{align*}
\]

we add 1 to the jump addresses in the original version of \( U \).

Furthermore, we assume that, if \( U \), it terminates always with the PC containing the number of the first instruction following it.

Means that after insertion, if, after having been inserted, \( U \) terminates, the next instruction to be executed is the one following \( U \).
In this Subsection we will introduce some higher level program constructs for URMs, and how to translate them back into the original URM language.

These constructs will be still be rather low level in terms of the theory of programming languages, but high enough in order to allow easily to introduce the programs needed in this module.

First of all we use the more readable statements

\[
\begin{align*}
R_k &:= R_k + 1 \quad \text{for succ}(k), \\
R_k &:= R_k \div 1 \quad \text{for pred}(k), \\
\text{if } R_k = 0 \text{ then goto } q & \quad \text{for ifzero}(k, q).
\end{align*}
\]
Labelled URM programs

- We introduce labelled URM programs.
- It will be easier to translate them back into original URM programs.
- The label End denotes the first instruction following a program.

So instead of

\[ I_0 = \text{if } R_0 = 0 \text{ then goto } 3 \]
\[ I_1 = R_0 := R_0 - 1 \]
\[ I_2 = \text{if } R_1 = 0 \text{ then goto } 0 \]

we write

\[ \text{LabelBegin: } I_0 = \text{if } R_0 = 0 \text{ then goto End} \]
\[ I_1 = R_0 := R_0 - 1 \]
\[ I_2 = \text{if } R_1 = 0 \text{ then goto LabelBegin} \]
Omitting $I_k =$

- We omit now “$I_k =$”.
- Furthermore, labels don’t have to start with Label, so we can write Begin instead of LabelBegin.
- We obtain the following program:

  Begin: if $R_0 = 0$ then goto End

  $R_0 := R_0 \div 1$

  if $R_1 = 0$ then goto Begin

  End:

- Since End : is always the first instruction following the program, we will omit the last line End :.
Replacing Registers by Variables

We write variable names instead of registers.
So if \( x, y \) denote \( R_0, R_1 \), respectively, we write instead of

\[
\text{Begin: } \quad \text{if } R_0 = 0 \text{ then goto End}
\]
\[
R_0 := R_0 \div 1
\]
\[
\text{if } R_1 = 0 \text{ then goto Begin}
\]

the following

\[
\text{Begin: } \quad \text{if } x = 0 \text{ then goto End}
\]
\[
x := x \div 1
\]
\[
\text{if } y = 0 \text{ then goto Begin}
\]
More Complex Statements

goto mylabel;
stands for the (labelled) URM statement
(aux denotes a new register):
if aux = 0 then goto mylabel;

If aux is not changed elsewhere, it contains always 0.
More Complex Statements

while $x \neq 0$ do {
    <Instructions>;
}

stands for the following URM program:

LabelLoop : if $x = 0$ then goto End;
            <Instructions>
            goto LabelLoop;
More Complex Statements

repeat{
    \langle Instructions \rangle
} until x = 0;

stands for the following URM program:

\langle Instructions \rangle;
while x \neq 0 do {
    \langle Instructions \rangle;
}

Note that this results in doubling of \langle Instructions \rangle.

One can avoid this.

But the length of the resulting program is not a problem as long as we are not dealing with complexity theory.
More Complex Statements

\[ x := 0 \]

stands for the following program:

\[
\text{while } x \neq 0 \text{ do } \{ x := x - 1; \};
\]
More Complex Statements

\[ y := x; \]

stands for (if \( x, y \) denote different registers, \( \text{aux} \) is new):

\[
\text{while } x \neq 0 \text{ do } \{
\hspace{1em} x := x \div 1;
\hspace{1em} \text{aux} := \text{aux} + 1; \}
\]

\[ y := 0; \]

\[
\text{while } \text{aux} \neq 0 \text{ do } \{
\hspace{1em} \text{aux} := \text{aux} \div 1;
\hspace{1em} x := x + 1;
\hspace{1em} y := y + 1; \}
\]

If \( x, y \) are the same register, \( y := x \) stands for the empty statement.
More Complex Statements

Assume $x$, $y$, $z$ denote different registers.

$x := y + z$; stands for the following program ($aux$ is an additional variable):

\[
\begin{align*}
x & := y; \\
aux & := z; \\
\text{while } aux \neq 0 \text{ do } \\
& \quad \{ \\
& \quad \quad aux := aux \cdot \frac{1}{2}; \\
& \quad \quad x := x + 1; \\
& \quad \}
\end{align*}
\]
More Complex Statements

Assume $x$, $y$, $z$ denote different registers. Remember, that $a \div b := \max\{0, a - b\}$.

$x := y \div z$;

is computed as follows ($aux$ is an additional variable):

$x := y$;

$aux := z$;

while $aux \neq 0$ do {
    $aux := aux \div 1$;
    $x := x \div 1$;
};
More Complex Statements

Assume $x$, $y$ denote different registers.

while $x \neq y$ do {
  \langle Statements \rangle;

stands for ($aux$, $aux_i$ denote new registers):

$aux_0 := x \div y$;

$aux_1 := y \div x$;

$aux := aux_0 + aux_1$;

while $aux \neq 0$ do {
  \langle Statements \rangle
  $aux_0 := x \div y$;
  $aux_1 := y \div x$;
  $aux := aux_0 + aux_1$;
};
We introduce some constructions for introducing URM-computable functions.

We will later introduce the set of partial recursive functions as the least set of functions closed under these constructions.

Then by the fact that the URM-computable functions are closed under these operations it follows that all partial recursive functions are URM-computable.

We introduce first names for all functions constructed this way.
Notations for Partial Functions

Definition 3.1

(a) Define the **zero function** $\text{zero} : \mathbb{N} \to \mathbb{N}$, $\text{zero}(x) = 0$.

(b) Define the **successor function** $\text{succ} : \mathbb{N} \to \mathbb{N}$,

$$\text{succ}(x) = x + 1.$$ 

(c) Define for $0 \leq i < n$ the **projection function**

$$\text{proj}_i^n : \mathbb{N}^n \to \mathbb{N}, \text{proj}_i^n(x_0, \ldots, x_{n-1}) = x_i.$$

Remark

- Note that all total functions are as well partial, so we have for instance as well $\text{zero} : \mathbb{N} \rightsquigarrow \mathbb{N}$.

- $\text{proj}_0^1 : \mathbb{N} \to \mathbb{N}$ is the identity function: $\text{proj}_0^1(x) = x$. 

CS_226 Computability Theory, Sec. 3 (c)
(d) Assume

\[ g : (B_0 \times \cdots \times B_{k-1}) \leadsto C, \]
\[ h_i : A_0 \times \cdots \times A_{n-1} \leadsto B_i \quad i = 0, \ldots, k - 1 \]

Define

\[ f := g \circ (h_0, \ldots, h_{k-1}) : A_0 \times \cdots \times A_{n-1} \leadsto C : \]

\[ f(\vec{a}) \leadsto g(h_0(\vec{a}), \ldots, h_{k-1}(\vec{a})) \]

In case of \( k = 1 \) we write \( g \circ h \) instead of \( g \circ (h) \).

Furthermore as usual

\[ g_1 \circ g_2 \circ \cdots \circ g_n := g_1 \circ (g_2 \circ (\cdots \circ (g_{n-1} \circ g_n))) \]
Notations for Partial Functions

(e) Assume

\[ g : \mathbb{N}^k \sim \rightarrow \mathbb{N} \, , \]
\[ h : \mathbb{N}^{k+2} \sim \rightarrow \mathbb{N} \, . \]

Then we can define a function \( f : \mathbb{N}^{k+1} \sim \rightarrow \mathbb{N} \) defined by
\textbf{primitive recursion} from \( g \) and \( h \) as follows:

\[
\begin{align*}
  f(\vec{n}, 0) & : \simeq g(\vec{n}) \\
  f(\vec{n}, m + 1) & : \simeq h(\vec{n}, m, f(\vec{n}, m))
\end{align*}
\]

We write \texttt{primrec}(g, h) for the function \( f \) just defined.

\[ \textbf{So} \; \texttt{primrec}(g, h) : \mathbb{N}^{k+1} \sim \rightarrow \mathbb{N}. \]
Notations for Partial Functions

In the special case $k = 0$, it doesn’t make sense to use $g()$. Instead replace in this case $g$ by some natural number. So the case $k = 0$ reads as follows:

Assume $n \in \mathbb{N}$, $h : \mathbb{N}^2 \leadsto \mathbb{N}$.

Define

$$f : \mathbb{N} \leadsto \mathbb{N}$$

by primitive recursion from $n$ and $h$ as follows:

$$f(0) :\simeq n$$
$$f(m + 1) :\simeq h(m, f(m))$$

We write $\text{primrec}(n, h)$ for $f$, so $\text{primrec}(n, h) : \mathbb{N} \leadsto \mathbb{N}$. 
Addition can be defined using primitive recursion:
Let \( \text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}, \text{add}(x, y) := x + y. \) We have

\[
\begin{align*}
\text{add}(x, 0) &= x + 0 = x \\
\text{add}(x, y + 1) &= x + (y + 1) = (x + y) + 1 = \text{add}(x, y) + 1
\end{align*}
\]

Therefore

\[
\begin{align*}
\text{add}(x, 0) &= g(x) \\
\text{add}(x, y + 1) &= h(x, y, \text{add}(x, y))
\end{align*}
\]

where

\[
\begin{align*}
g : \mathbb{N} \rightarrow \mathbb{N}, & \quad g(x) := x, \\
h : \mathbb{N}^3 \rightarrow \mathbb{N}, & \quad h(x, y, z) := z + 1.
\end{align*}
\]

So \( \text{add} = \text{primrec}(g, h). \)
Addition (add)

\[ g : \mathbb{N} \rightarrow \mathbb{N} , \quad g(x) := x , \]
\[ h : \mathbb{N}^3 \rightarrow \mathbb{N} , \quad h(x, y, z) := z + 1 , \]

\text{add} := \text{primrec}(g, h)

We have

\[ \text{add}(x, 0) = g(x) = x = x + 0. \]
\[ \text{add}(x, 1) = h(x, 0, \text{add}(x, 0)) = \text{add}(x, 0) + 1 = x + 1. \]
\[ \text{add}(x, 2) = h(x, 1, \text{add}(x, 1)) = \text{add}(x, 1) + 1 = (x + 1) + 1. \]

etc.
Examples for Primitive Recursion

Multiplication can be defined using primitive recursion:
Let \( \text{mult} : \mathbb{N}^2 \rightarrow \mathbb{N} \), \( \text{mult}(x, y) := x \cdot y \). We have

\[
\text{mult}(x, 0) = x \cdot 0 = 0 \\
\text{mult}(x, y + 1) = x \cdot (y + 1) = x \cdot y + x = \text{mult}(x, y) + x
\]

Therefore

\[
\text{mult}(x, 0) = g(x) \\
\text{mult}(x, y + 1) = h(x, y, \text{mult}(x, y))
\]

where

\[
g : \mathbb{N} \rightarrow \mathbb{N} , \quad g(x) := 0 , \\
h : \mathbb{N}^3 \rightarrow \mathbb{N} , \quad h(x, y, z) := z + x.
\]

So \( \text{mult} = \text{primrec}(g, h) \).
Multiplication \((\text{mult})\)

\[
g : \mathbb{N} \rightarrow \mathbb{N} , \quad g(x) := 0 ,
\]
\[
h : \mathbb{N}^3 \rightarrow \mathbb{N} , \quad h(x, y, z) := z + x ,
\]
\[
\text{mult} := \text{primrec}(g, h)
\]

We have

\[
\text{mult}(x, 0) = g(x) = 0 = x \cdot 0.
\]
\[
\text{mult}(x, 1) = h(x, 0, \text{mult}(x, 0)) = \text{mult}(x, 0) + x = 0 + x = x.
\]
\[
\text{mult}(x, 2) = h(x, 1, \text{mult}(x, 1)) = \text{mult}(x, 1) + x = (x \cdot 1) + x.
\]

etc.
Examples for Primitive Recursion

Let \( \text{pred} : \mathbb{N} \rightarrow \mathbb{N}, \text{pred}(n) := n - 1 = \begin{cases} n - 1 & \text{if } n > 0, \\ 0 & \text{otherwise.} \end{cases} \)

\text{pred} can be defined using primitive recursion:

\[
\begin{align*}
\text{pred(0)} & = 0 \\
\text{pred}(x + 1) & = x 
\end{align*}
\]

Therefore

\[
\begin{align*}
\text{pred(0)} & = 0 \\
\text{pred}(x + 1) & = h(x, \text{pred}(x)) 
\end{align*}
\]

where

\[
h : \mathbb{N}^2 \rightarrow \mathbb{N}, \quad h(x, y) := x
\]

So \( \text{pred} = \text{primrec}(0, h) \).

CS_226 Computability Theory, Sec. 3 (c)
Examples for Primitive Recursion

• \( x \div y \) can be defined using primitive recursion:

Let \( f(x, y) := x \div y \). We have

\[
\begin{align*}
  f(x, 0) &= x \div 0 = x \\
  f(x, y + 1) &= x \div (y + 1) = (x \div y) \div 1 \\
                 &= \text{pred}(x \div y) = \text{pred}(f(x, y))
\end{align*}
\]

Therefore

\[
\begin{align*}
  f(x, 0) &= g(x) \\
  f(x, y + 1) &= h(x, y, f(x, y))
\end{align*}
\]

where

\[
\begin{align*}
  g : \mathbb{N} \to \mathbb{N} \ , \quad g(x) &:= x \\
  h : \mathbb{N}^3 \to \mathbb{N} \ , \quad h(x, y, z) &:= \text{pred}(z)
\end{align*}
\]

So \( f = \text{primrec}(g, h) \).
Remark

We have

\[ f(\vec{n}, m) \uparrow \rightarrow \forall k \geq m. f(\vec{n}, k) \uparrow \]

**Proof:**

- We have

\[ f(\vec{n}, m + 1) :\simeq h(\vec{n}, m, f(\vec{n}, m)) \]

- All functions are strict.
- So if \( f(\vec{n}, m) \uparrow \), then

\[ f(\vec{n}, m + 1) \simeq h(\vec{n}, m, f(\vec{n}, m)) \uparrow \]

therefore

\[ f(\vec{n}, m + 1) \uparrow \]
Proof of Remark

Therefore we have

\[ f(\vec{n}, m) \uparrow \rightarrow f(\vec{n}, m + 1) \uparrow . \]

By induction it follows that \( f(\vec{n}, m) \uparrow \) implies

\[ \forall k \geq m. f(\vec{n}, k) \uparrow . \]
Example

Let

\[ h : \mathbb{N}^2 \rightarrow \mathbb{N} , \quad h(n, m) \equiv \begin{cases} m-1 & \text{if } m > 0, \\ \perp & \text{otherwise.} \end{cases} \]

Let

\[ f : \mathbb{N} \rightarrow \mathbb{N} , \quad f := \text{primrec}(1, h) , \]

i.e.

\[ f(0) \equiv 1 , \quad f(n + 1) \equiv h(n, f(n)) . \]

Then

\[ f(0) \equiv 1 \]
\[ f(1) \equiv h(0, f(0)) \equiv h(0, 1) \equiv 0 \]
\[ f(2) \equiv h(1, f(1)) \equiv h(1, 0) \uparrow \]
\[ \forall m \geq 2. f(m) \uparrow \]
Notations for Partial Functions

Let \( g : \mathbb{N}^{n+1} \rightarrow \mathbb{N} \).
We define \( \mu y.g(\vec{x}, y) \simeq 0 : = \)

\[
(\mu y.g(\vec{x}, y) \simeq 0) : = \begin{cases}
\text{the least } y \in \mathbb{N} \text{ s.t.} & \\
g(\vec{x}, y) \simeq 0 & \\
\text{and for } 0 \leq y' < y & \\
\text{there exists a } z' \neq 0 & \\
s.t. g(\vec{x}, y') \simeq z' & \text{if such } y \\
\bot & \text{otherwise}
\end{cases}
\]
Now define \( h : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}, \)

\[
h(\vec{x}) \sim (\mu y.g(\vec{x}, y) \sim 0)
\]

We write \( \mu(g) \) for this function \( h. \)
Examples

Assume

\[ g(x, 0) \simeq 1 \]
\[ g(x, 1) \uparrow \]
\[ g(x, 2) \simeq 0 \]

Then

\[ (\mu y. g(x, y) \simeq 0) \uparrow \]

Assume instead

\[ g(x, 0) \simeq 1 \]
\[ g(x, 1) \simeq 5 \]
\[ g(x, 2) \simeq 0 \]

Then

\[ (\mu y. g(x, y) \simeq 0) \simeq 2 \]
Computation of $\mu(g)$

$\mu(g)(\vec{x}) := \mu y. (g(\vec{x}, y) \simeq 0)$.

If $g$ is intuitively computable, we see that $h := \mu(g)$ is intuitively computable as follows:

- In order to compute $h(\vec{x})$ we first compute $g(\vec{x}, 0)$.
  - If this computation never terminates $g(\vec{x}, 0) \uparrow$ and $(\mu y. g(\vec{x}, y) \simeq 0) \uparrow$ as well.
  - If it terminates, and we have $g(\vec{x}, 0) \simeq 0$, we obtain $(\mu y. g(\vec{x}, y) \simeq 0) \simeq 0$.

- Otherwise, repeat the above with testing of $g(\vec{x}, 1) \simeq 0$.
  - If successful $(\mu y. g(\vec{x}, y) \simeq 0) \simeq 1$.
  - If unsuccessful repeat it with 2, 3, etc.
Computation of $\mu(g)$

Note that $\mu(g)(\vec{x}) \uparrow$ in case there is a $y$ s.t.

$$g(\vec{x}, y) \uparrow$$

and for $y' < y$ we have $g(\vec{x}, y') \downarrow$ but $g(\vec{x}, y') \simeq z$ for some $z > 0$.

This coincides with computation by the above mentioned intuitive computation:

In this case, the program will compute $g(\vec{x}, 0)$, $g(\vec{x}, 1), \ldots, g(\vec{x}, y - 1)$ and get as result that these values are $\neq 0$.

Then it will try to compute $g(\vec{x}, y)$, and this computation never terminates.

So the value of this program is undefined, as is $(\mu g)(\vec{x})$. 
Computation of $\mu(g)$

If we defined $\mu(g)(\bar{x})$ to be the least $z$ s.t.

$$g(\bar{x}, y) \simeq 0$$

independently of whether $g(\bar{x}, y') \downarrow$ for all $y' < y$, then we would obtain a non computable function.
Examples for $\mu$

1. Let $f : \mathbb{N}^2 \to \mathbb{N}$, $f(x, y) := x \div y$. Then

   $$(\mu y. f(x, y) \simeq 0) \simeq x$$

   so $\mu(f)(x) \simeq x$.

2. Let $f : \mathbb{N} \leadsto \mathbb{N}$,
   $f(0) \uparrow$,
   $f(n) := 0$ for $n > 0$.

   Then

   $$(\mu y. f(y) \simeq 0) \uparrow$$

   .
Examples for $\mu$

Let $f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$,

$$f(n) := \begin{cases} 1 & \text{if there exist primes } p, q < 2n + 4 \\ 0 & \text{otherwise} \end{cases} \text{ s.t. } 2n + 4 = p + q,$$

$\mu y. f(y) \approx 0$ is the first $n$ s.t. there don’t exist primes $p, q$ s.t. $2n + 4 = p + q$.

Goldbach’s conjecture says that every even number $\geq 4$ is the sum of two primes. This is equivalent to $(\mu y. f(y) \approx 0) \uparrow$.

It is one of the most important open problems in mathematics to show (or refute) Goldbach’s conjecture. If we could decide whether a partial computing function is defined (which we can’t), we could decide Goldbach’s conjecture.
Next Step

We are going to show that the URM computable functions are closed under the operations introduced above.

In order to show this we need to be able to modify URM programs, so that they
- have some other specified input and output registers,
- and conserve the content of certain other registers.

The following lemma shows that such a modification is possible.
Lemma and Definition 3.2

Assume $f : \mathbb{N}^k \to \mathbb{N}$ is URM-computable. Assume $x_0, \ldots, x_{k-1}, y, z_0, \ldots, z_l$ are different variables. Then one can define a URM program, which, computes $f(x_0, \ldots, x_{k-1})$ and stores the result in $y$ in the following sense:

- If $f(x_0, \ldots, x_{k-1}) \downarrow$, the program terminates at the first instruction following this program, and stores the result in $y$.

- If $f(x_0, \ldots, x_{k-1}) \uparrow$, the program never terminates.

The program can be defined so that it doesn’t change $x_0, \ldots, x_{k-1}, z_0, \ldots, z_l$.

For $U$ we say it is a URM program which computes $y \sim f(x_0, \ldots, x_{k-1})$ and avoids $z_0, \ldots, z_l$. 
Intuition behind Lem. 3.2

Lemma 3.2 means that if $f$ is URM-computable then we can define a URM-program in such a way that it takes the arguments from registers we have chosen, and stores the result in a register we have chosen, and does this in such a way that the content of the input registers and of some other registers we have chosen are not modified. This is possible as long as the input registers and the output register are all different.
Idea of the proof

- First copy the arguments in some other registers, so that the arguments are preserved.
- Then compute the function on those auxiliary registers and make sure that the computation doesn’t affect the registers to be preserved.
- Then move the result into the register chosen as output register.

Omit Proof.
Proof

Let \( U \) be a URM program s.t. \( U^{(k)} = f \).

Let \( u_0, \ldots, u_{k-1} \) be registers different from the above.

By renumbering of registers and of jump addresses, we obtain a program \( U' \), which computes the result of \( f(u_0, \ldots, u_{k-1}) \) in \( u_0 \).

leaves the registers mentioned in the lemma unchanged, and which, if it terminates, terminates in the first instruction following \( U' \).

The following is a program as intended:

\[
\begin{align*}
    u_0 & := x_0; \\
    \ldots \\
    u_{k-1} & := x_{k-1}; \\
    U' & \\
    y & := u_0;
\end{align*}
\]
Lemma 3.3

(a) zero, succ and proj\textsubscript{i}\textsuperscript{n} are URM-computable.

(b) If f : \mathbb{N}^n \rightarrow \mathbb{N}, g_i : \mathbb{N}^k \rightarrow \mathbb{N} are URM-computable, so is f \circ (g_0, \ldots, g_{n-1}).

(c) If g : \mathbb{N}^n \rightarrow \mathbb{N}, and h : \mathbb{N}^{n+2} \rightarrow \mathbb{N} are URM-computable, so is the function f := \text{primrec}(g, h) defined by primitive recursion from g and h.

(d) If g : \mathbb{N}^{n+1} \rightarrow \mathbb{N} is URM-computable, so is \mu(g).
Remark

The Lemma is very powerful:

- It shows that many functions are URM-computable.
- This shows that for instance the exponential function is URM computable.
- This follows since addition, multiplication and exponentiation can be defined by primitive recursion from the basic functions.
- Writing a URM program directly which computes the exponential function would be very difficult.
Proof of Lemma 3.3 (a)

Let \( x_i \) denote register \( R_i \).

**Proof of (a)**

- **zero** is computed by the following program:
  \[
  x_0 := 0.
  \]

- **succ** is computed by the following program:
  \[
  x_0 := x_0 + 1.
  \]

- **proj\(_k\)^n** is computed by the following program:
  \[
  x_0 := x_k.
  \]

Especially, if \( k = 0 \) then \( \text{proj}\(_k^n\) \) is the empty program (i.e. the program with no instructions this is since we defined \( x_0 := x_0 \) to be the empty program.)
Proof of Lemma 3.3 (b)

Assume \( f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N} \), \( g_i : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N} \) are URM-computable. Show \( f \circ (g_0, \ldots, g_{n-1}) \) is computable.

A plan for the program is as follows:

- Input is stored in registers \( x_0, \ldots, x_{k-1} \).
  - Let \( \bar{x} := x_0, \ldots, x_{k-1} \).
- First we compute \( g_i(\bar{x}) \) for \( i = 0, \ldots, n - 1 \), store result in registers \( y_i \).
  - By Lemma 3.2 we can do this in such a way that \( x_0, \ldots, x_{k-1} \) and the previously computed values \( g_i(\bar{x}) \), which are stored in \( y_j \) for \( j < i \) are not destroyed.
- Then compute \( f(y_0, \ldots, y_{n-1}) \), and store result in \( x_0 \).
- Then \( x_0 \) contains \( f(g_0(\bar{x}), \ldots, g_{n-1}(\bar{x})) \).
Proof of Lemma 3.3 (b)

Let therefore $U_i$ be a URM program $(i = 0, \ldots, n - 1)$, which computes $y_i \simeq g_i(\bar{x})$ and avoids $y_j$ for $j \neq i$.

Let $V$ be a URM program, which computes $x_0 \simeq f(y_0, \ldots, y_{n-1})$. 
Proof of Lemma 3.3 (b)

Let $U'$ be defined as follows:

$U_0$

$\ldots$

$U_{n-1}$

$V$

We show $U'^{(k)}(\bar{x}) \simeq (f \circ (g_0(\bar{x}), \ldots, g_{n-1}(\bar{x})))$.

Omit rest of proof.
Proof of Lemma 3.3 (b)

$U'$ is the program

$U_0$

$\ldots$

$U_{n-1}$

$V$

- **Case 1:** For one $i \ g_i(\bar{x}) \uparrow$.
  The program will loop in program $U_i$ for the first such $i$.

\[ U^{(k)}(\bar{x}) \uparrow, \ f \circ (g_0, \ldots, g_{n-1})(\bar{x}) \uparrow. \]

- **Case 2:** For all $i \ g_i(\bar{x}) \downarrow$.
  The program executes $U_i$, sets $y_i \simeq g_i(x_0, \ldots, x_{k-1})$ and reaches beginning of $V$. 

CS_226 Computability Theory, Sec. 3 (c)
Proof of Lemma 3.3 (b)

$U'$ is the program

$U_0$

$\ldots$

$U_{n-1}$

$V$

- **Case 2.1:** $f(g_0(\overline{x}), \ldots, g_{n-1}(\overline{x})) \uparrow$. $V$ will loop, $U'(k)(\overline{x}) \uparrow$, $f \circ (g_0, \ldots, g_{n-1})(\overline{x}) \uparrow$.

- **Case 2.2:** Otherwise. The program reaches the end of program $V$ and result in $x_0 \simeq f(g_0(\overline{x}), \ldots, g_{n-1}(\overline{x}))$. So $U'(k)(\overline{x}) \simeq (f \circ (g_0, \ldots, g_{n-1}))(\overline{x})$. 
Proof of Lemma 3.3 (b)

In all cases

\[ U^{(k)}(\bar{x}) \simeq (f \circ (g_0, \ldots, g_{n-1}))(\bar{x}) \]
Proof of Lemma 3.3 (c)

Assume
\[ g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}, \quad h : \mathbb{N}^{n+2} \xrightarrow{\sim} \mathbb{N} \]
are URM-computable.

Let
\[ f := \text{primrec}(g, h) . \]

Show \( f \) is URM-computable.

Defining equations for \( f \) are as follows
(let \( \vec{n} := n_0, \ldots, n_{n-1} \):

\[ f(\vec{n}, 0) \preceq g(\vec{n}), \]
\[ f(\vec{n}, k + 1) \preceq h(\vec{n}, k, f(\vec{n}, k)). \]
Proof of Lemma 3.3 (c)

Computation of $f(\vec{n}, l)$ for $l > 0$ is as follows:

- Compute $f(\vec{n}, 0)$ as $g(\vec{n})$.
- Compute $f(\vec{n}, 1)$ as $h(\vec{n}, 0, f(\vec{n}, 0))$, using the previous result.
- Compute $f(\vec{n}, 2)$ as $h(\vec{n}, 1, f(\vec{n}, 1))$, using the previous result.
- ... 
- Compute $f(\vec{n}, l)$ as $h(\vec{n}, l - 1, f(\vec{n}, l - 1))$, using the previous result.
Proof of Lemma 3.3 (c)

Plan for the program:

Let $\bar{x} := x_0, \ldots, x_{n-1}$.
Let $y, z, u$ be new registers.

Compute $f(\bar{x}, y)$ for $y = 0, 1, 2, \ldots, x_n$, and store result in $z$.

Initially we have $y = 0$ (holds for all registers except of $x_0, \ldots, x_n$ initially).
We compute $z \simeq g(\bar{x}) (\simeq f(\bar{x}, 0))$.
Then $y = 0, z \simeq f(\bar{x}, 0)$. 
Proof of Lemma 3.3 (c)

In step from $y$ to $y + 1$:

- Assume that we have $z \simeq f(\bar{x}, y)$.
- We want that after increasing $y$ by 1 the **loop invariant** $z \simeq f(\bar{x}, y)$ still holds.

Obtained as follows

- Compute $u \simeq h(\bar{x}, y, z)$
  
  $(\simeq h(\bar{x}, y, f(\bar{x}, y)) \simeq f(\bar{x}, y + 1))$.

- Execute $z := u \simeq f(\bar{x}, y + 1)$.

- Execute $y := y + 1$.

- At the end, $z \simeq f(\bar{x}, y)$ for the new value of $y$.

Repeat this until $y = x_n$.

Once $y$ has reached $x_n$, $z$ contains $f(\bar{x}, y) \simeq f(\bar{x}, x_n)$.

Execute $x_0 := z$. 
Proof of Lemma 3.3 (c)

Let

- $U$ be a URM program, which computes $z \simeq g(\vec{x})$ and avoids $y$ (by definition 3.2, it doesn’t modify the arguments $\vec{x}$ of $g$);

- $V$ be a program, which computes $u \simeq h(\vec{x}, y, z)$. (by definition 3.2, it doesn’t change $\vec{x}, y, z$.)
Proof of Lemma 3.3 (c)

Let $U'$ be as follows:

\[
\begin{align*}
U & \quad \text{% Compute } z \simeq g(\vec{x}) (\simeq f(\vec{x}, 0)) \\
\text{while } x_n \neq y \text{ do } \{ & \\
V & \quad \text{% Compute } u \simeq h(\vec{x}, y, z) \\
& \quad \text{% will be } \simeq h(\vec{x}, y, f(\vec{x}, y)) \simeq f(\vec{x}, y + 1) \\
& \quad \text{z := u;} \\
& \quad y := y + 1; \} \\
x_0 := z;
\end{align*}
\]
Proof of Lemma 3.3 (c)

Correctness of this program:

- When $U$ has terminated, we have $y = 0$ and $z \simeq g(\bar{x}) \simeq f(\bar{x}, y)$.

- After each iteration of the while loop, we have $y := y' + 1$ and $z \simeq h(\bar{x}, y', z')$.
  $(y', z' \text{ are the previous values of } y, z, \text{ respectively.})$

- Therefore we have $z \simeq f(\bar{x}, y)$.

- The loop terminates, when $y$ has reached $x_n$.
  Then $z$ contains $f(\bar{x}, y)$.
  This is stored in $x_0$. 

Proof of Lemma 3.3 (c)

If $U$ loops for ever, or in one of the iterations $V$ loops for ever, then:

- $U'$ loops, $U'(n+1)(\vec{x}, x_n) \uparrow$.
- $f(\vec{x}, k) \uparrow$ for some $k < x_n$,
- subsequently $f(\vec{x}, l) \uparrow$ for all $l > k$.
- Especially, $f(\vec{x}, x_n) \uparrow$.
- Therefore $f(\vec{x}, x_n) \simeq U'(n+1)(\vec{x}, x_n)$.
Proof of Lemma 3.3 (d)

Assume

\[ g : \mathbb{N}^{n+1} \rightarrow \mathbb{N} \]

is URM-computable.

Show

\[ \mu(g) \]

is URM-computable as well.

Note \( \mu(g)(x_0, \ldots, x_{k-1}) \) is the minimal \( z \) s.t.

\[ g(x_0, \ldots, x_{k-1}, z) \approx 0. \]

Let \( \vec{x} := x_0, \ldots, x_{k-1} \) and let \( y, z \) be registers different from \( \vec{x} \).
Proof of Lemma 3.3 (d)

Plan for the program:

- **Compute** $g(\overline{x}, 0) , g(\overline{x}, 1) , \ldots$ until we find a $k$ s.t. $g(\overline{x}, k) \simeq 0$.
  Then return $k$.

- **This is carried out by executing**

  $$z \simeq g(\overline{x}, y)$$

  and successively increasing $y$ by 1 until we have $z = 0$. 
Proof of Lemma 3.3 (d)

Let $U$ compute

$$z \simeq g(x_0, \ldots, x_{k-1}, y),$$

(and preserve the arguments $x_0, \ldots, x_{k-1}, y$.) Let $V$ be as follows:

repeat{
    U
    $y := y + 1$;
} until ($z = 0$);

$y := y - 1$;

$x_0 := y$;

Omit rest of proof.
Proof of Lemma 3.3 (d)

\begin{align*}
V \text{ is} \quad & \text{repeat}\{U; y := y + 1; \} \text{ until } (z = 0); \\
y := y - 1; x_0 := y;
\end{align*}

Initially $y = 0$.

After each iteration of the repeat loop, we have

\[ y := y' + 1 \, , \, z \simeq g(x_0, \ldots, x_{k-1}, y') \]

(y' is the value of $y$ before this iteration).

If the loop terminates, we have

\[ z \simeq 0 \, , \, y = y' + 1 \]

where $y'$ is the first value, such that $g(x_0, \ldots, x_{k-1}, y') \simeq 0$. 

Finally $y$ is decreased by one.

Then $y$ is the least $y$ s.t.

$$g(x_0, \ldots, x_{k-1}, y) \simeq 0.$$ 

$x_0$ is then set to that value.
(d) Configurations of URMs

We will later simulate URMs by Turing machines.

For this we need the notation of a configuration of a URM.

Describes the state of a URM during one of the intermediate steps.

This state is determined if we know

- the value of the PC,
- the content of $R_0, \ldots, R_N$, if $R_N$ is the maximum register referenced to in the URM program.

- Note that this $N$ can be computed from a URM program.
Definition 3.4

Let $U = I_0, \ldots, I_n$ be a URM-program.

Let $N$ be the maximum number $l$ s.t. an instruction $\text{succ}(l)$, $\text{pred}(l)$, $\text{ifzero}(l, q)$ is amongst $I_0, \ldots, I_n$.

Then a configuration for this $U$ is given as

$$\pi(k, \langle n_0, \ldots, n_N \rangle)$$

s.t. $k, n_0, \ldots, n_N \in \mathbb{N}$.

This configuration represents the state of an execution of a URM where

- PC contains $k$,
- register $R_i$ contains $n_i$ for $i = 0, \ldots, N$,
- and all other registers contain 0.
Definition 3.4

In case of the empty URM program, we set in the definition above $N = -1$, i.e. a configuration is of the form

$$\pi(k, \langle \rangle)$$

for some $k \in \mathbb{N}$.

Note that from a configuration of a URM and the URM program we can intuitively compute

- whether the URM has terminated,
- and, in case it has not terminated, the configuration after the next instruction of the URM has been executed.
Undecidability of the Halting Problem first proved 1936 by Alan Turing.

We present this in context of URM\textsuperscript{s}.

In the following “computable” means “URM-computable”.

Justified later 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 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 **decidable**, if the **characteristic function** of \( M \) defined by

\[
\chi_M(\vec{x}) := \begin{cases} 
1 & \text{if } M(\vec{x}) \text{ holds,} \\
0 & \text{otherwise}
\end{cases}
\]

is computable.
Characteristic function

If we treat true as 1 and false as 0, then the characteristic function is nothing but the Boolean valued function which decides whether $M(\overline{x})$ holds or not:

$$
\chi_M(\overline{x}) = \begin{cases} 
\text{true} & \text{if } M(\overline{x}) \text{ holds,} \\
\text{false} & \text{otherwise}
\end{cases}
$$
Example of URM-Comp. Functions

The binary predicate

\[ \text{Multiple}(x, y) :\iff x \text{ is a multiple of } y \]

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:

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

\( \chi_{\text{Multiple}} \) is intuitively computable, therefore \( \text{Multiple} \) is decidable.
URM programs

• URM-programs can be written as a string of ASCII-symbols.

• → URM programs elements of $A^*$, where $A =$ set of ASCII-symbols.

• → URM programs can be encoded as natural numbers. Let for a URM program $U$, $\text{encode}(U) \in \mathbb{N}$ be its code.

• It is intuitively decidable, whether a string of ASCII symbols is a URM-program.
  
  • One can show that this can be decided by a URM.

• → It is intuitively decidable, whether $n = \text{encode}(U)$ for a URM-program $U$.

• However, the halting problem is undecidable.
Definition 3.6

The Halting Problem is the following binary predicate:

\[
\text{Halt}(x, y) \iff \begin{cases} 
1 & \text{if } x = \text{encode}(U) \text{ for a URM program } U \\
& \text{and } U^{(1)}(y) \downarrow \\
0 & \text{otherwise}
\end{cases}
\]
Example

- Let $e = \text{encode}(U)$, where $U$ is the URM program $U$

  $I_0 = \text{ifzero}(0, 0)$

- If input is $> 0$, the program terminates immediately, and $R_0$ remains unchanged, so

  $U^{(0)}(k) \simeq k$

  for $k > 0$.

- If input is $= 0$, the program loops for ever. Therefore

  $U^{(0)}(0) \uparrow$.

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

Below we will see: \texttt{Halt} is undecideable.

However, \texttt{WeakHalt} is computable:

\[
\text{WeakHalt}(x, y) : \simeq \begin{cases} 
1 & \text{if } x = \text{encode}(U) \\
& \text{for a URM program } U \\
& \text{and } U^{(1)}(y) \downarrow \\
\bot & \text{otherwise}
\end{cases}
\]

Computed as follows:
First check whether \( x = \text{encode}(U) \) for some URM program \( U \).
If not, enter an infinite loop.
Otherwise, simulate \( U \) with input \( y \).
If simulation stops, output 1, otherwise the program loops for ever.
Question

What is $\text{WeakHalt}(e, n)$ where $e$ is a code for the URM program

$I_0 = \text{ifzero}(0, 0)$?
Theorem 3.7

The halting problem is undecidable.

Proof:
Assume $U^{(2)}$ decides the Halting Problem, i.e.

$$U^{(2)}(x, y) \sim \begin{cases} 1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

We define a computable function $f : \mathbb{N} \leadsto \mathbb{N}$, which cannot be computed by the URM with code $e$, as violated on input $e$.

We argue similar to the proof that $\mathbb{N} \not\sim \mathcal{P}(\mathbb{N})$:
Proof of Theorem 3.7

\[
U^{(2)}(x, y) \simeq \begin{cases} 
1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\
0 & \text{otherwise.}
\end{cases}
\]

If \( U^{(2)}(e, e) \simeq 1 \), i.e. \( e \) encodes a URM \( V \) and \( V^{(1)}(e) \downarrow \), then let \( f(e) \uparrow \).

Therefore, \( V^{(1)} \neq f \).
Proof of Theorem 3.7

\[ U^{(2)}(x, y) \simeq \begin{cases} 
1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\
0 & \text{otherwise.} 
\end{cases} \]

If \( U^{(2)}(e, e) \simeq 0 \), i.e. \( e \) doesn’t encode a URM, or it encodes a URM \( V \) and \( V^{(1)}(e) \uparrow \), then we let \( f(e) \downarrow: f(e) :\simeq 0 \).

Therefore, if \( e \) encodes a URM \( V \), we have \( V^{(1)} \neq f \).

The complete definition is as follows:

\[ f(e) \simeq \begin{cases} 
0 & \text{if } U^{(2)}(e, e) \simeq 0 \\
\perp & \text{otherwise.} 
\end{cases} \]
Proof of Theorem 3.7

\[ U^{(2)}(x, y) \simeq \begin{cases} 
1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\
0 & \text{otherwise.} 
\end{cases} \]

\[ f(e) \simeq \begin{cases} 
0 & \text{if } U^{(2)}(e,e) \simeq 0 \\
\bot & \text{otherwise.} 
\end{cases} \]

- \( f \) is not URM-computable:
  - If \( f \) were URM computable, \( f = V^{(1)} \) for some \( V \).
  - Then \( f = V^{(1)} \) is violated by \( f(\text{encode}(V)) \neq V^{(1)}(\text{encode}(V)) \).
Proof of Theorem 3.7

\[ U^{(2)}(x, y) \simeq \begin{cases} 
1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\
0 & \text{otherwise.}
\end{cases} \]

\[ f(e) \simeq \begin{cases} 
0 & \text{if } U^{(2)}(e, e) \simeq 0 \\
\bot & \text{otherwise.}
\end{cases} \]

In short the argument is as follows: Assume \( f = V^{(1)} \). Then

\[ V^{(1)}(\text{encode}(V)) \downarrow \quad \text{property of } U \quad \Leftrightarrow \quad U^{(2)}(\text{encode}(V), \text{encode}(V)) \simeq 1 \]

Def of \( f \)

\[ f = V^{(1)} \quad \Leftrightarrow \quad f(\text{encode}(V)) \uparrow \]

\[ V^{(1)}(\text{encode}(V)) \uparrow \]
Proof of Theorem 3.7

\[ U^{(2)}(x, y) \simeq \begin{cases} 
1 & \text{if } x \text{ encodes a URM program } V \text{ s.t. } V^{(1)}(y) \downarrow \\
0 & \text{otherwise.} 
\end{cases} \]

\[ f(e) \simeq \begin{cases} 
0 & \text{if } U^{(2)}(e, e) \simeq 0 \\
\bot & \text{otherwise.} 
\end{cases} \]

Therefore, \( f \neq V^{(1)} \), \( f \) cannot be computed by any URM program \( V \). However, \( f \) can be intuitively computed, (the program makes use of \( U \)). It can be shown that it is in fact URM-computable. So we get a contradiction.
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.