(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 that 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.

---

(b) **Higher level programming concepts for URMs.**

---

(c) **URM computable functions.**

---

**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.

---

**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.
The URM

- 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.

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.

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.

URM

John Shepherdson (Bristol) (2nd from the right)
Developed together with Sturgis the URM.
The URM

- **URM Instructions**
  - 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.
      - → 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
      \[ R_k := R_k - 1 \]
$x - y$

Here

$$x - y := \max\{x - y, 0\},$$

i.e.

$$x - y = \begin{cases} x - y & \text{if } y \leq x, \\ 0 & \text{otherwise}. \end{cases}$$

---

**URM Instructions**

The **conditional jump instruction**

```
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
  ```

---

**Example of a URM Program**

The following is an example of a URM-program:

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

---

**Finiteness**

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

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

If we run this program with initial values \( R_0 = 2 \), \( R_1 = 0 \), we obtain the following trace of a run of this program:

\[ \begin{array}{c|c|c|c}
    \text{Instruction} & R_0 & R_1 \\
    \hline
    I_0 & 2 & 0 \\
\end{array} \]
Example

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

If we run this program with initial values \( R_0 = 2, R_1 = 0 \), we obtain the following trace of a run of this program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( R_0 )</th>
<th>( R_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>I_1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>I_2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

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

If we run this program with initial values \( R_0 = 2, R_1 = 0 \), we obtain the following trace of a run of this program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( R_0 )</th>
<th>( R_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>I_1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>I_2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I_0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I_1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

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

If we run this program with initial values \( R_0 = 2, R_1 = 0 \), we obtain the following trace of a run of this program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( R_0 )</th>
<th>( R_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_0 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

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

If we run this program with initial values \( R_0 = 2, R_1 = 0 \), we obtain the following trace of a run of this program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( R_0 )</th>
<th>( R_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_0 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( I_3 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

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

If we run this program with initial values \( R_0 = 2, R_1 = 0 \), we obtain the following trace of a run of this program:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( R_0 )</th>
<th>( R_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_0 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( I_2 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( I_0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( I_3 )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

URM Stops
Behaviour of the Example

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

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 \( R_0 := 0 \).

URM-Computable Functions

For every U-program we define the function defined by it.

In fact there are many functions 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 \( U = I_0, \ldots, I_{n-1} \) 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
    \[ U^{(k)}(a_0, \ldots, a_{k-1}) \simeq b \]

- If the program never stops,
  \[ U^{(k)}(a_0, \ldots, a_{k-1}) \uparrow \]
**URM-Computable Functions**

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

**Example**

Consider the example of a URM-program treated before:

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

We have seen that if \( R_1 \) is initially zero, then the program reduces \( R_0 \) to 0 and then stops.

**Change of Notation**

Until the academic year 2004/05, \( P \) 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 2004/05 and before.

**Example**

A computation of \( U^{(1)}(k) \) is as follows:

- We set \( R_0 \) to \( k \), all other registers to 0.
- Then the URM program is executed, starting with instruction \( I_0 \).
- This program terminates, with \( R_0 \) containing 0.
- The value returned is the content of \( R_0 \), i.e. 0.
- Therefore \( U^{(1)}(k) \simeq 0 \).
Example

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

- In order to compute \( U^{(2)}(k, l) \) we have to do the same, but set initially \( R_0 \) to \( k \), \( R_1 \) to \( l \).
- For \( l = 0 \) we obtain the same run of the URM program as before.
  - Therefore \( U^{(2)}(k, 0) \simeq 0 \).
- What is \( U^{(2)}(k, l) \) for \( l > 0 \)?

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. Function

The following function is computable:

\[ f : \mathbb{N}^2 \rightarrow \mathbb{N} , \quad f(x, y) \simeq x + y \]

We derive a URM-program for it in several steps.

**Step 1:**
Initially \( R_0 \) contains \( x \), \( R_1 \) contains \( y \), and the other registers contain 0.
Program should then terminate with \( R_0 \) containing \( f(x, y) \), i.e. \( x + y \).
A higher level program is as follows:

\[ R_0 := R_0 + R_1 \]
Example of URM-Comp. Function

\[ R_0 := R_0 + R_1 \]

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

\[
\text{while } R_1 \neq 0 \text{ do }
\begin{aligned}
R_0 &:= R_0 + 1 \\
R_1 &:= R_1 - 1
\end{aligned}
\]

- This increases \( R_0 \) by 1 as many times as the value contained in \( R_1 \).
- This means that the content of \( R_1 \) is added to \( R_0 \).
- Note that at the end of the run, \( R_1 \) contains 0. But this is no problem since at the end we only read off the result from \( R_0 \), and ignore \( R_1 \).

**Example of URM-Comp. Function**

\[
\text{while } R_1 \neq 0 \text{ do }
\begin{aligned}
R_0 &:= R_0 + 1 \\
R_1 &:= R_1 - 1
\end{aligned}
\]

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

\[
\text{LabelBegin: if } R_1 = 0 \text{ then goto LabelEnd;}
\begin{aligned}
R_0 &:= R_0 + 1; \\
R_1 &:= R_1 - 1;
\end{aligned}
\text{if } R_2 = 0 \text{ then goto LabelBegin;}
\]

**Example of URM-Comp. Function**

\[
\text{LabelBegin: if } R_1 = 0 \text{ then goto LabelEnd;}
\begin{aligned}
R_0 &:= R_0 + 1; \\
R_1 &:= R_1 - 1;
\end{aligned}
\text{if } R_2 = 0 \text{ then goto LabelBegin;}
\]

**Step 4:**
Replace last goto by a conditional goto, depending on \( R_2 = 0 \).
\( R_2 \) is initially 0 and never modified, therefore this jump will always be carried out.

**Example of URM-Comp. Function**

\[
\text{LabelBegin: if } R_1 = 0 \text{ then goto LabelEnd;}
\begin{aligned}
R_0 &:= R_0 + 1; \\
R_1 &:= R_1 - 1;
\end{aligned}
\text{if } R_2 = 0 \text{ then goto LabelBegin;}
\]

**Step 5:**
Translate the program into a URM program \( I_0, I_1, I_2, I_3 \):

\[
I_0 = \text{ifzero}(1,4) \\
I_1 = \text{succ}(0) \\
I_2 = \text{pred}(1) \\
I_3 = \text{ifzero}(2,0)
\]
(b) **High Level Progr. Constructs**

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.

---

**Convention Concerning Jump Addresses**

- When inserting URM programs $U$ as part of new URM programs, jump addresses will be adapted accordingly.

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

  we add 1 to the jump addresses in the original version of $U$.

  Furthermore, we assume that, if $U$ terminates, it terminates with the PC containing the number of the first instruction following $U$.

  Means that if we then insert $U$, and a run of $U$ terminates, the next instruction to be executed is the one following $U$.

---

**More Readable Statements**

- We use the more readable statements

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

---

**Labelled URM programs**

- We introduce labelled URM programs.

- It will be easier to translate them back into original URM programs.

- The label $\text{End}$ denotes the first instruction following a program.

- So instead of

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

  we write

  \[
  \begin{align*}
  \text{LabelBegin} : \\
  I_0 &= \text{if } R_0 = 0 \text{ then goto End} \\
  I_1 &= R_0 := R_0 - 1 \\
  I_2 &= \text{if } R_1 = 0 \text{ then goto LabelBegin}
  \end{align*}
  \]

  \[
  \text{End}:
  \]

---
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:

  \[
  \begin{align*}
  \text{Begin} & : \quad \text{if } R_0 = 0 \text{ then goto End} \\
  & \quad R_0 := R_0 - 1 \\
  & \quad \text{if } R_1 = 0 \text{ then goto Begin} \\
  \text{End} & :
  \end{align*}
  \]
- 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

\[
\begin{align*}
\text{Begin} & : \quad \text{if } R_0 = 0 \text{ then goto End} \\
& \quad R_0 := R_0 - 1 \\
& \quad \text{if } R_1 = 0 \text{ then goto Begin} \\
\text{End} & :
\end{align*}
\]

Goto

- \text{goto mylabel;} stands for the (labelled) URM statement
  \[
  \text{if aux0} = 0 \text{ then goto mylabel;} 
  \]
- Here aux0 is a register (which we can keep fixed), which is initially zero and never modified in the URM program, so it contains always 0.

while $x \neq 0$ do \{ \ldots \}

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

stands for the following URM program:

\[
\begin{align*}
\text{LabelLoop} & : \quad \text{if } x = 0 \text{ then goto End} \\
& \quad \{Instructions\} \\
& \quad \text{goto LabelLoop;}
\end{align*}
\]
Repeat Loop

repeat{
    ⟨Instructions⟩
} until x = 0;

stands for the following URM program:

⟨Instructions⟩;
while x ≠ 0 do {
    ⟨Instructions⟩;
}

Note that this results in doubling of ⟨Instructions⟩.

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.

x := 0

stands for the following program:

while x ≠ 0 do {x := x + 1;};

y := x;

y := x;
stands for (if x, y denote different registers, aux is new):

aux := 0
while x ≠ 0 do {
    x := x + 1;
    aux := aux + 1;};
y := 0;
while aux ≠ 0 do {
    aux := aux − 1;
    x := x + 1;
    y := y + 1;};

If x, y are the same register, y := x stands for the empty statement.

y := x;

Note that the URM program y := x; preserved the value of x.

So after executing the URM program, x contains the value as it had before starting the execution.

Similarly, in the URM programs introduced on the next slides

x := y + z
x := y − z

the values of y and z will preserved.
Checking for Inequality

We have

\[(x \div y) + (y \div x) \neq 0 \iff x \neq y\]

Proof:

- If \(x > y\), then
  
  \[
  \begin{align*}
  x \div y &> 0 , \\
  y \div x &= 0 , \\
  (x \div y) + (y \div x) &> 0 
  \end{align*}
  \]

- If \(y > x\), then
  
  \[
  \begin{align*}
  y \div x &> 0 , \\
  x \div y &= 0 , \\
  (x \div y) + (y \div x) &> 0 
  \end{align*}
  \]

So a while loop

\[
\text{while } x \neq y \text{ do } \{ \ldots \}
\]

can be replaced by

\[
\text{while } (x \div y) + (y \div x) \neq 0 \text{ do } \{ \ldots \}
\]

if we unfold this, we obtain the following:

Checking for Inequality

\[
(x \div y) + (y \div x) \neq 0 \iff x \neq y
\]

- If \(x = y\), then
  
  \[
  \begin{align*}
  y \div x &= 0 , \\
  x \div y &= 0 , \\
  (x \div y) + (y \div x) &= 0 
  \end{align*}
  \]

Assume \(x, y, z\) denote different registers.

\[
x := y + z;
\]

Assume \(x, y, z\) denote different registers.

\[
x := y + z; \text{ stands for the following program } (\text{aux} \text{ is an additional variable)}:
\]

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

Assume \(x, y, z\) denote different registers.

\[
x := y - z;
\]

Assume \(x, y, z\) denote different registers.

Remember, that \(a \div b := \max\{0, a - b\}\).

\[
x := y - z;
\]

is computed as follows (\(\text{aux} \text{ is an additional variable)}:

\[
x := y; \\
\text{aux} := z; \\
\text{while } \text{aux } \neq 0 \text{ do } \{ \\
\quad \text{aux} := \text{aux } \div 1; \\
\quad x := x - 1; \};
\]
while \( x \neq y \) do \{ \cdots \}

Assume \( x, y \) denote different registers.

while \( x \neq y \) do {
    \(<Statements>\);  
stands for (\( \text{aux}, \text{aux}_i \) denote new registers):

\[
\begin{align*}
\text{aux}_0 &:= x - y; \\
\text{aux}_1 &:= y - x; \\
\text{aux} &:= \text{aux}_0 + \text{aux}_1;
\end{align*}
\]
while \( \text{aux} \neq 0 \) do {
    \(<Statements>\)  
    \[
    \begin{align*}
    \text{aux}_0 &:= x - y; \\
    \text{aux}_1 &:= y - x; \\
    \text{aux} &:= \text{aux}_0 + \text{aux}_1;
    \end{align*}
    \]
}

\[CS_226\] Computability Theory, Lent Term 2008, Sec. 3 (b) 3-49

\section*{(c) URM-Computable Functions}

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.

\[CS_226\] Computability Theory, Lent Term 2008, Sec. 3 (c) 3-50

\section*{Notations for Partial Functions}

\textbf{Definition 3.1}

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

(b) Define the \textbf{successor function} \( \text{succ} : \mathbb{N} \to \mathbb{N}, \text{succ}(x) = x + 1 \).

(c) Define for \( 0 \leq i < n \) the \textbf{projection function} \( \text{proj}_n^i : \mathbb{N}^n \to \mathbb{N}, \text{proj}_n^i(x_0, \ldots, x_{n-1}) = x_i \).

\textbf{Remark}

- Note that all total functions are as well partial, so we have for instance as well \( \text{zero} : \mathbb{N} \sim \mathbb{N} \).
- \( \text{proj}_0^1 : \mathbb{N} \to \mathbb{N} \) is the identity function: \( \text{proj}_0^1(x) = x \).

\[CS_226\] Computability Theory, Lent Term 2008, Sec. 3 (c) 3-51

\section*{Notations for Partial Functions}

(d) Assume

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

Define

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

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

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

Furthermore as usual

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

\[CS_226\] Computability Theory, Lent Term 2008, Sec. 3 (c) 3-52
Notations for Partial Functions

(e) Assume

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

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

\[ f(\bar{n}, 0) : \simeq g(\bar{n}) \]
\[ f(\bar{n}, m + 1) : \simeq h(\bar{n}, m, f(\bar{n}, m)) \]

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

So \( \text{primrec}(g, h) : \mathbb{N}^{k+1} \rightarrow \mathbb{N} \).

primrec in Haskell

In Haskell we can define \( \text{primrec} \) as a higher-order function as follows:

```
data Nat = Z | S Nat
 deriving Show
```

- - \( \text{primrec0} \) is the operator for primitive recursion
- - defining a 1-ary function \( \text{primrec0} f a :: \text{Nat} \rightarrow \text{Nat} \)
- - from \( f : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \) and \( a : \text{Nat} \)

\[ \text{primrec0} :: \text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat} \]
\[ \text{primrec0} a g \text{Z} = a \]
\[ \text{primrec0} a g (S n) = g n (\text{primrec0} a g n) \]

primrec in Haskell (Cont.)

- - \( \text{primrec1} \) is the operator for primitive recursion
- - defining a 2-ary function \( \text{primrec1} f g :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \)
- - from \( f : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \) and \( g : \text{Nat} \rightarrow \text{Nat} \)

\[ \text{primrec1} :: (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \]
\[ \text{primrec1} g h n Z = g n \]
\[ \text{primrec1} g h n (S m) = h n m (\text{primrec1} g h n m) \]
Examples for Primitive Recursion

Addition can be defined using primitive recursion:
Let \( add : \mathbb{N}^2 \to \mathbb{N}, add(x, y) := x + y \). We have

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

Therefore

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

where

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

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

Defining \( + \) from primrec in Haskell

In Haskell we can define \( add \) from \( \text{primrec} \) as follows

\[
\text{add} :: \text{Nat} \to \text{Nat} \to \text{Nat}
\]

\[
\text{add} = \text{primrec1} (\lambda n \to n) (\lambda n m k \to \text{S} k)
\]

Examples for Primitive Recursion

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

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

Therefore

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

where

\[
\begin{align*}
g : \mathbb{N} \to \mathbb{N}, & \quad g(x) := 0 \\
h : \mathbb{N}^3 \to \mathbb{N}, & \quad h(x, y, z) := z + x
\end{align*}
\]

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

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

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

Examples for Primitive Recursion

Let $\text{pred} : \mathbb{N} \to \mathbb{N},$
\[ \text{pred}(n) := n - 1 = \begin{cases} n - 1 & \text{if } n > 0, \\ 0 & \text{otherwise}. \end{cases} \]
\[ \text{pred} \] can be defined using primitive recursion:
\[ \text{pred}(0) = 0 \]
\[ \text{pred}(x + 1) = x \]
Therefore
\[ \text{pred}(0) = 0 \]
\[ \text{pred}(x + 1) = h(x, \text{pred}(x)) \]
where
\[ h : \mathbb{N}^2 \to \mathbb{N}, \quad h(x, y) := x \]
\[ \text{pred} = \text{primrec}(0, h). \]

Examples for Primitive Recursion

$x \div y$ can be defined using primitive recursion:
Let $f(x, y) := x \div y$. We have
\[ f(x, 0) = x \div 0 = x \]
\[ f(x, y + 1) = x \div (y + 1) = (x \div y) - 1 \]
\[ = \text{pred}(x \div y) = \text{pred}(f(x, y)) \]
Therefore
\[ f(x, 0) = g(x) \]
\[ f(x, y + 1) = h(x, y, f(x, y)) \]
where
\[ g : \mathbb{N} \to \mathbb{N}, \quad g(x) := x, \]
\[ h : \mathbb{N}^3 \to \mathbb{N}, \quad h(x, y, z) := \text{pred}(z). \]
So $f = \text{primrec}(g, h)$.

Remark

If $f = \text{primrec}(g, h)$, then
\[ f(\bar{n}, m) \uparrow \to \forall k \geq m. f(\bar{n}, k) \uparrow \]

Proof:
We have
\[ f(\bar{n}, m + 1) \simeq h(\bar{n}, m, f(\bar{n}, m)) \]
All functions are strict.
So if $f(\bar{n}, m) \uparrow$, then
\[ f(\bar{n}, m + 1) \simeq h(\bar{n}, m, f(\bar{n}, m)) \uparrow \]
therefore
\[ f(\bar{n}, m + 1) \uparrow. \]
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

\[ h : \mathbb{N}^2 \cong \mathbb{N}, \quad h(n, m) = \begin{cases} m + 1 & \text{if } m > 0, \\ \bot & \text{otherwise}. \end{cases} \]

Let

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

i.e.

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

Then

\[
\begin{align*}
    f(0) & \cong 1 \\
    f(1) & \cong h(0, f(0)) \cong h(0, 1) \cong 0 \\
    f(2) & \cong h(1, f(1)) \cong h(1, 0) \\
    \forall m \geq 2. f(m) & \uparrow
\end{align*}
\]

Primitive-Recursive Functions

The functions, which can be defined from \( \text{zero}, \text{succ}, \) \( \text{proj}_i^k \) by using composition (\( \circ \)) and primitive recursion (\( \text{primrec} \)) are called the \textit{primitive recursive functions}.

The primitive-recursive functions will be studied more in detail in Sect. 5.

There we will see that they are powerful, but \textit{not Turing-complete}.

Notations for Partial Functions

Let \( g : \mathbb{N}^{n+1} \cong \mathbb{N}. \)

We define \( \mu y.g(\vec{x}, y) \cong 0 :\)

\[ (\mu y.g(\vec{x}, y) \cong 0) : \]

the least \( y \in \mathbb{N} \) s.t.

\[ g(\vec{x}, y) \cong 0 \]

and for \( 0 \leq y' < y \)

there exists a \( z' \neq 0 \)

s.t. \( g(\vec{x}, y') \cong z' \) if such \( y \)

exists,

\[ \bot \]

otherwise
Now define \( h : \mathbb{N}^n \to \mathbb{N} \),

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

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

### Examples

Assume

\[
\begin{align*}
  g(x, 0) & \simeq 1 \\
  g(x, 1) & \uparrow \\
  g(x, 2) & \simeq 0
\end{align*}
\]

Then

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

Assume instead

\[
\begin{align*}
  g(x, 0) & \simeq 1 \\
  g(x, 1) & \simeq 5 \\
  g(x, 2) & \simeq 0
\end{align*}
\]

Then

\[ (\mu y. g(\vec{x}, y) \simeq 0) \simeq 2 \]

### Computation of \( \mu(g) \)

\[ \mu(g)(\vec{x}) :\simeq \mu y. (g(\vec{x}, y) \simeq 0) \].

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.

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 \( g(\vec{x}, 0) \), \( g(\vec{x}, 1) \), …, \( g(\vec{x}, y - 1) \) 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 $\mu(g)(\vec{x})$ to be the least $z$ s.t.

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

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

Examples for $\mu$

Let $f : \mathbb{N} \rightarrow \mathbb{N}$,

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

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

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.

Partial Recursive Functions

The functions, which can define in the same way as the primitive-recursive functions

- i.e. being defined from zero, $\text{succ}$, $\text{proj}_k^i$ by using composition ($\circ$) and primitive recursion ($\text{primrec}$) but by additionally closing them under $\mu$, are called the **partial recursive functions**.

The partial recursive functions will be studied more in detail in Sect. 6.

There we will see that the partial recursive functions form a Turing complete model of computation.
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.

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.

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 \( y \approx f(x_0, \ldots, x_{k-1}) \) and preserves \( z_0, \ldots, z_l \).

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:

$$u_0 := x_0;$$
$$\ldots$$
$$u_{k-1} := x_{k-1};$$
$$U'$$
$$y := u_0;$$

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.

Omit Proof.

Lemma 3.3

(a) zero, succ and $\text{proj}_{n}^{k}$ 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 $\mu(g)$.

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: $x_0 := x_0 + 1$.
- $\text{proj}_{n}^{k}$ is computed by the following program: $x_0 := x_k$.

Especially, if $k = 0$ then $\text{proj}_{n}^{k}$ 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 \( f : \mathbb{N}^n \rightarrow \mathbb{N}, g_i : \mathbb{N}^k \rightarrow \mathbb{N} \) 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 \( \vec{x} := x_0, \ldots, x_{k-1} \).
- First we compute \( g_i(\vec{x}) \) 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 \( g_i(\vec{x}) \), 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 \( f(g_0(\vec{x}), \ldots, g_{n-1}(\vec{x})) \).

Proof of Lemma 3.3 (b)

Let \( U' \) be defined as follows:
\[
U_0 \ldots U_{n-1} V
\]
We show \( U'^{(k)}(\vec{x}) \simeq (f \circ (g_0(\vec{x}), \ldots, g_{n-1}(\vec{x}))) \).
Omit rest of proof.
Proof of Lemma 3.3 (b)

In all cases

\[ U'(x) \simeq (f \circ (g_0, \ldots, g_{n-1}))(x) \]

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 \( \vec{n} := n_0, \ldots, n_{n-1} \):

\[ f(\vec{n}, 0) \simeq g(\vec{n}), \]

\[ f(\vec{n}, k + 1) \simeq h(\vec{n}, k, f(\vec{n}, k)). \]

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.
- \( \ldots \)
- 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 \( \vec{x} := x_0, \ldots, x_{n-1} \).
- Let \( y, z, u \) be new registers.
- Compute \( f(\vec{x}, y) \) 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 \( z \approx g(\vec{x}) \approx f(\vec{x}, 0) \).
  - Then \( y = 0, z \approx f(\vec{x}, 0) \).

In step from \( y \) to \( y + 1 \):

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

  Obtained as follows
  - Compute \( u \approx h(\vec{x}, y, z) \)
    \( (\approx h(\vec{x}, y, f(\vec{x}, y)) \approx f(\vec{x}, y + 1)) \).
  - Execute \( z := u \approx f(\vec{x}, y + 1) \).
  - Execute \( y := y + 1 \).
  - At the end, \( z \approx f(\vec{x}, y) \) for the new value of \( y \).  

Repeat this until \( y = x_n \).

- Once \( y \) has reached \( x_n \), \( z \) contains \( f(\vec{x}, y) \approx f(\vec{x}, x_n) \).
- Execute \( x_0 := z \).

Proof of Lemma 3.3 (c)

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

- \( V \) be a program, which computes \( u \approx h(\vec{x}, y, z) \). (by definition 3.2, it doesn’t change \( \vec{x}, y, z \).)

Let \( U' \) be as follows:

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

Correctness of this program:

- When $U$ has terminated, we have $y = 0$ and $z \simeq g(\vec{x}) \simeq f(\vec{x}, y)$.
- After each iteration of the while loop, we have $y := y' + 1$ and $z \simeq h(\vec{x}, y', z')$.
  ($y', z'$ are the previous values of $y, z$, respectively.)
- Therefore we have $z \simeq f(\vec{x}, y)$.
- The loop terminates, when $y$ has reached $x_n$.
  Then $z$ contains $f(\vec{x}, y)$.
  This is stored in $x_0$.

Proof of Lemma 3.3 (d)

Plan for the program:

- Compute $g(\vec{x}, 0), g(\vec{x}, 1), \ldots$ until we find a $k$ s.t. $g(\vec{x}, k) \simeq 0$.
  Then return $k$.
- This is carried out by executing
  
  $z \simeq g(\vec{x}, y)$

  and successively increasing $y$ by 1 until we have $z = 0$.

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.

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

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)

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:

\[
    \text{repeat}\{
        U \\
        y := y + 1;
    \} \quad \text{until} \; (z = 0); \\
    y := y - 1; \\
    x_0 := y;
\]

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.

Proof of Lemma 3.3 (d)

Initially \( y = 0 \).

After each iteration of the repeat loop, we have

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

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

If the loop terminates, we have

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

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