Sect. 3: The URM

(a) Definition of the URM.
(b) Higher level programming concepts for URMs.
(c) URM computable functions.

(a) Definition of the URM

- A **model of computation** consists of a set of partial computable functions together with methods, which describe, how to compute those functions.
  - One aims at models of computation which are **complete**.
    - Here a model of computation is **complete**, if it contains all computable functions.
  - Since “intuitively computable” is not a mathematical notion, completeness is not a mathematical notion and cannot be proved mathematically.

Turing Completeness

- Sometimes by “complete” it is meant that the model contains all functions computable by a Turing machine – then one obtains a mathematical definition.
- We use **Turing complete** for this mathematical definition.
  - So a model is Turing complete if it contains all functions computable by a Turing machine.

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 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.

URM

John Shepherdson (Bristol) (2nd from the right)

Developed together with Sturgis the URM.

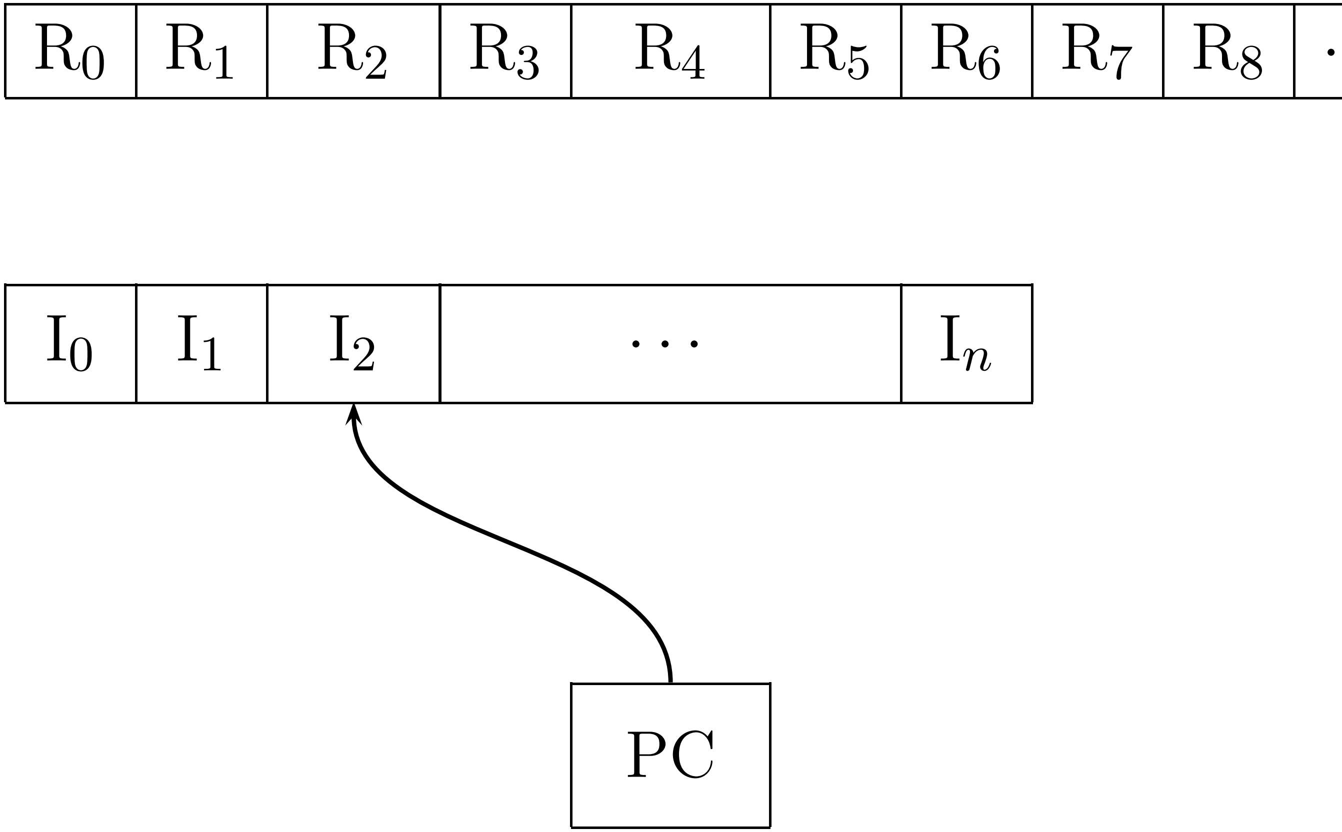
Description of the URM

- The **URM** consists of
  - infinitely many registers R_i
    - can store arbitrarily big natural number;
  - a **URM program** consisting of a finite sequence of **instructions** $\mathrm{I}_0, \mathrm{I}_1, \mathrm{I}_2, \ldots I_n$;
  - 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.

Remark

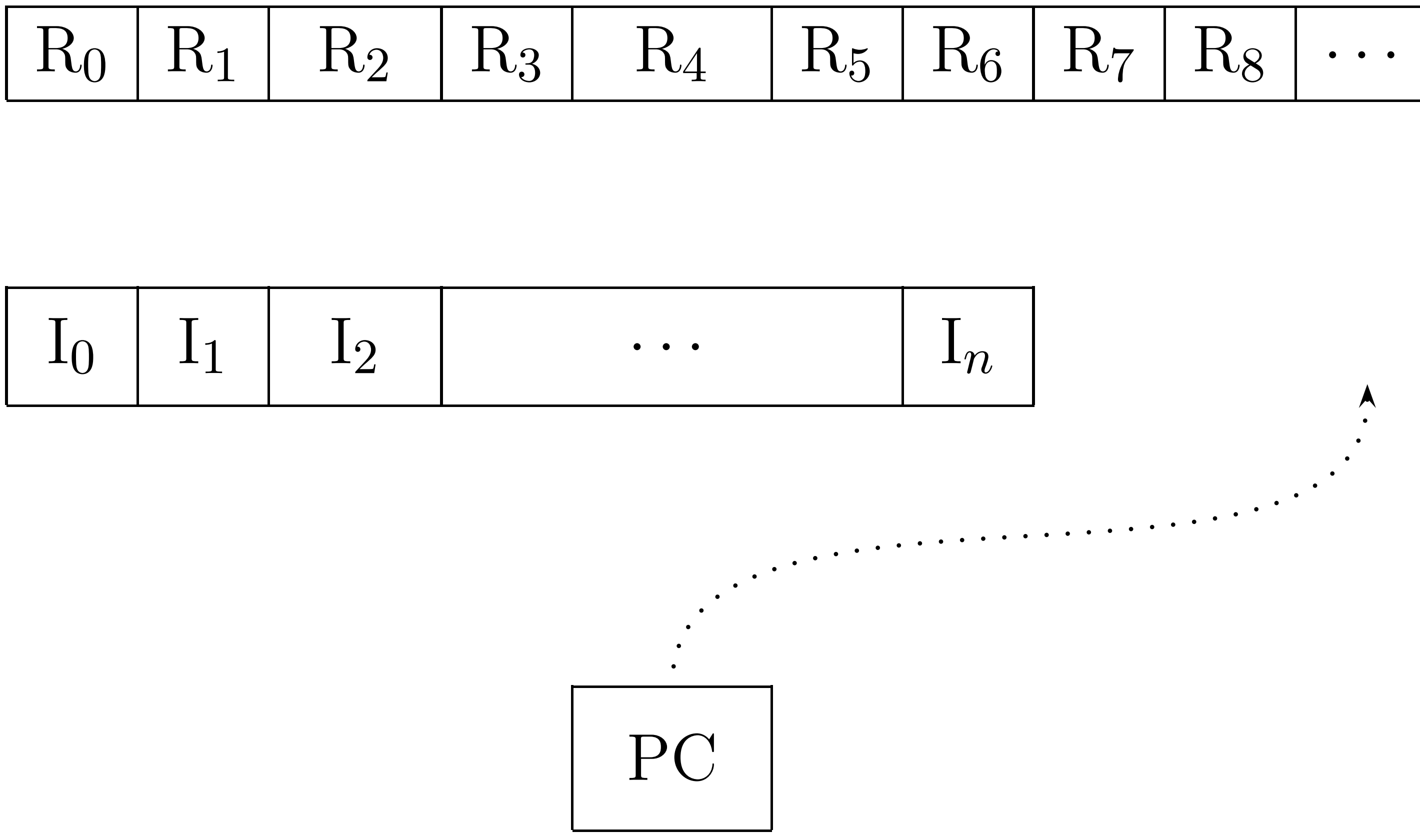
- Note that the URM program is part of the URM.
- One could distinguish between
  - The architecture of a URM consisting of registers, the program counter and a memory for a URM program,
  - and the URM program itself.
- For historic reasons by a URM we mean the URM architecture **together** with a URM program.

The URM



Execute Instruction

The URM



Program has terminated

URM Instructions

- 3 kinds of **URM instructions**.
  - The **successor instruction**

    $$\mathtt{succ}(k) \quad ,$$

    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

    $$\mathrm{R}_k := \mathrm{R}_k + 1$$

URM Instructions

- The **predecessor instruction**

$$\texttt{pred}(k) \quad ,$$

  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

$$\mathrm{R}_k := \mathrm{R}_k \dot{-} 1$$

$x \dot{-} y$

- Here

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

i.e.

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

URM Instructions

- The **conditional jump instruction**

  $$\texttt{ifzero}(k, q)$$

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

  - If R_k contains 0, PC is set to q

    $\rightarrow$ 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

  $$\texttt{if } \mathrm{R}_k = 0 \texttt{ then goto } q$$

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:

$$\begin{aligned} \mathrm{I}_0 &= \ \mathtt{ifzero}(0,3) \\ \mathrm{I}_1 &= \ \mathtt{pred}(0) \\ \mathrm{I}_2 &= \ \mathtt{ifzero}(1,0) \end{aligned}$$

Example

$\mathrm{I}_0 = \mathtt{ifzero}(0,3) \qquad \mathrm{I}_1 = \mathtt{pred}(0) \qquad \mathrm{I}_2 = \mathtt{ifzero}(1,0)$

Example

$\mathrm{I}_0 = \mathtt{ifzero}(0, 3)$ $\mathrm{I}_1 = \mathtt{pred}(0)$ $\mathrm{I}_2 = \mathtt{ifzero}(1, 0)$

If we run this program with inital values $\mathrm{R}_0 = 2$, $\mathrm{R}_1 = 0$, we obtain the following trace of a run of this program:

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0

Example

$\mathrm{I}_0 = \mathtt{ifzero}(0, 3)$ $\mathrm{I}_1 = \mathtt{pred}(0)$ $\mathrm{I}_2 = \mathtt{ifzero}(1, 0)$

If we run this program with inital values $\mathrm{R}_0 = 2$, $\mathrm{R}_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0

Example

$\mathrm{I}_0 = \mathtt{ifzero}(0,3)$ $\mathrm{I}_1 = \mathtt{pred}(0)$ $\mathrm{I}_2 = \mathtt{ifzero}(1,0)$

If we run this program with inital values $\mathrm{R}_0 = 2$, $\mathrm{R}_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0
I_1	1	0

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0
I_1	1	0
I_2	0	0

Example

$\mathrm{I}_0 = \texttt{ifzero}(0,3)$ $\mathrm{I}_1 = \texttt{pred}(0)$ $\mathrm{I}_2 = \texttt{ifzero}(1,0)$

If we run this program with inital values $\mathrm{R}_0 = 2$, $\mathrm{R}_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0
I_1	1	0
I_2	0	0
I_0	0	0

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0
I_1	1	0
I_2	0	0
I_0	0	0
I_3	0	0

Example

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

If we run this program with inital values $R_0 = 2$, $R_1 = 0$, we obtain the following trace of a run of this program:

Instruction	R_0	R_1
I_0	2	0
I_1	2	0
I_2	1	0
I_0	1	0
I_1	1	0
I_2	0	0
I_0	0	0
I_3	0	0

URM Stops

Behaviour of the Example

$$\begin{aligned} I_0 &= \texttt{ifzero}(0,3) \\ I_1 &= \texttt{pred}(0) \\ I_2 &= \texttt{ifzero}(1,0) \end{aligned}$$

- 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 corresonds 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 function which are defined by the same U-program:
  - A unary function $\mathrm{U}^{(1)}$, 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 undefinded.
  - A binary function $\mathrm{U}^{(2)}$, which stores its two arguments in R_0 and R_1, then operates as $\mathrm{U}^{(1)}$.
  - And so on. In general we obtain a k-ary partial function $\mathrm{U}^{(k)}$ for every $k \geq 1$.

Definition $\mathrm{U}^{(k)}$

- Let $\mathrm{U} = \mathrm{I}_0, \ldots, \mathrm{I}_{n-1}$ be a URM program, $k \in \mathbb{N}, k \geq 1$.
- We define a function

$$\mathrm{U}^{(k)} : \mathbb{N}^k \overset{\sim}{\to} \mathbb{N}$$

by determining how it is computed:

- Assume we want to compute $\mathrm{U}^{(k)}(a_0, \ldots, a_{k-1})$.
- **Initialisation:**
  - PC set to 0.
  - $a_0, \ldots, a_{k-1}$ stored in registers $\mathrm{R}_0, \ldots, \mathrm{R}_{k-1}$, respectively.
  - All other registers set to 0.
    (Sufficient to do this for registers referenced in the program).

URM-Computable Functions

- **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

$$\mathrm{U}^{(k)}(a_0, \ldots, a_{k-1}) \simeq b \quad .$$

  - If the program never stops,

$$\mathrm{U}^{(k)}(a_0, \ldots, a_{k-1})\uparrow \quad .$$

URM-Computable Functions

- $f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ is **URM-computable**, if $f = \mathrm{U}^{(k)}$ for some $k \in \mathbb{N}$ and some URM program U.

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

- Consider the example of a URM-program treated before:

$$\begin{aligned} \mathrm{I}_0 &= \texttt{ifzero}(0,3) \\ \mathrm{I}_1 &= \texttt{pred}(0) \\ \mathrm{I}_2 &= \texttt{ifzero}(1,0) \end{aligned}$$

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

Example

$$
\begin{aligned}
\mathrm{I}_0 &= \texttt{ifzero}(0,3)\\
\mathrm{I}_1 &= \texttt{pred}(0)\\
\mathrm{I}_2 &= \texttt{ifzero}(1,0)
\end{aligned}
$$

- A computation of $\mathrm{U}^{(1)}(k)$ 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 $\mathrm{U}^{(1)}(k) \simeq 0$.

Example

$$\begin{aligned} I_0 &= \texttt{ifzero}(0,3) \\ I_1 &= \texttt{pred}(0) \\ I_2 &= \texttt{ifzero}(1,0) \end{aligned}$$

- In order to compute $\mathrm{U}^{(2)}(k,l)$ 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 $\mathrm{U}^{(2)}(k,0) \simeq 0$.
- What is $\mathrm{U}^{(2)}(k,l)$ for $l > 0$?

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 infinitally 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. Function

The following function is computable:

$$f : \mathbb{N}^2 \overset{\sim}{\to} \mathbb{N} \ , \qquad f(x, y) \simeq x + y$$

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:

$$\mathrm{R}_0 := \mathrm{R}_0 + \mathrm{R}_1$$

Example of URM-Comp. Function

$R_0 := R_0 + R_1$

Step 2:

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

```
while $(R_1 \neq 0)$ do {$R_0 := R_0 + 1$
                         $R_1 := R_1 \dot{-} 1$}
```

- 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 the at the end we only read off the result from R_0, and ignore R_1.

Example of URM-Comp. Function

$$\texttt{while}\ (R_1 \neq 0)\ \texttt{do}\ \{R_0 := R_0 + 1 \\ R_1 := R_1 \dot{-} 1\}$$

Step 3:

Replace the while-loop by a goto:

```
LabelBegin :  if $R_1 = 0$ then goto LabelEnd;
              $R_0 := R_0 + 1$;
              $R_1 := R_1 \dot{-} 1$;
              goto LabelBegin;
LabelEnd :
```

Example of URM-Comp. Function

```
LabelBegin :  if $R_1 = 0$ then goto LabelEnd;
              $R_0 := R_0 + 1; R_1 := R_1 \dot{-} 1;$ goto LabelBegin;
LabelEnd :
```

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.

```
LabelBegin :  if $R_1 = 0$ then goto LabelEnd;
              $R_0 := R_0 + 1;$
              $R_1 := R_1 \dot{-} 1;$
              if $R_2 = 0$ then goto LabelBegin;
LabelEnd :
```

Example of URM-Comp. Function

```
LabelBegin :  if $R_1 = 0$ then goto LabelEnd;
              $R_0 := R_0 + 1$;
              $R_1 := R_1 \dot{-} 1$;
              if $R_2 = 0$ then goto LabelBegin;
LabelEnd :
```

Step 5:

Resolve labels:

```
0 :  if $R_1 = 0$ then goto 4;
1 :  $R_0 := R_0 + 1$;
2 :  $R_1 := R_1 \dot{-} 1$;
3 :  if $R_2 = 0$ then goto 0;
4 :
```

Example of URM-Comp. Function

```
0 : if $R_1 = 0$ then goto 4;
1 : $R_0 := R_0 + 1$;
2 : $R_1 := R_1 \dot{-} 1$;
3 : if $R_2 = 0$ then goto 0;
4 :
```

Step 6:

Translate the program into a URM program I_0, I_1, I_2, I_3:

$$\begin{aligned} I_0 &= \mathtt{ifzero}(1,4) \\ I_1 &= \mathtt{succ}(0) \\ I_2 &= \mathtt{pred}(1) \\ I_3 &= \mathtt{ifzero}(2,0) \end{aligned}$$

(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 Addr

- When inserting URM programs U as part of new URM programs, jump addresses will be adapted accordingly.
- E.g.in $\mathtt{succ}(0)$

  U

  $\mathtt{pred}(0)$

  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

$$\begin{array}{lll} \mathrm{R}_k := \mathrm{R}_k + 1 & \text{for} & \texttt{succ}(k), \\ \mathrm{R}_k := \mathrm{R}_k \dot{-} 1 & \text{for} & \texttt{pred}(k), \\ \texttt{if}\ \mathrm{R}_k = 0\ \texttt{then}\ \texttt{goto}\ q & \text{for} & \texttt{ifzero}(k, q). \end{array}$$

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 $$\begin{aligned} I_0 &= \ \texttt{if}\ R_0 = 0\ \texttt{then goto}\ 3 \\ I_1 &= \ R_0 := R_0 \dot{-} 1 \\ I_2 &= \ \texttt{if}\ R_1 = 0\ \texttt{then goto}\ 0 \end{aligned}$$
- we write

$$\begin{aligned} \texttt{LabelBegin}: \quad I_0 &= \ \texttt{if}\ R_0 = 0\ \texttt{then goto End} \\ I_1 &= \ R_0 := R_0 \dot{-} 1 \\ I_2 &= \ \texttt{if}\ R_1 = 0\ \texttt{then goto LabelBegin} \\ \texttt{End}: \quad & \end{aligned}$$

Omitting $\mathrm{I}_k =$

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

$$\begin{array}{ll}
\texttt{Begin}: & \texttt{if}\ \mathrm{R}_0 = 0\ \texttt{then goto End} \\
 & \mathrm{R}_0 := \mathrm{R}_0 \mathbin{\dot{-}} 1 \\
 & \texttt{if}\ \mathrm{R}_1 = 0\ \texttt{then goto Begin} \\
\texttt{End}: &
\end{array}$$

- Since $\texttt{End}:$ is always the first instruction following the program, we will omit the last line $\texttt{End}:$.

Replacing Registers by Variables

We write variable names instead of registers.
So if $\mathtt{x}$, $\mathtt{y}$ denote R_0, R_1, respectively, we write instead of

$$\begin{array}{ll}\mathtt{Begin}: & \mathtt{if}\ \mathrm{R}_0 = 0\ \mathtt{then\ goto\ End} \\ & \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ & \mathtt{if}\ \mathrm{R}_1 = 0\ \mathtt{then\ goto\ Begin}\end{array}$$

the following

$$\begin{array}{ll}\mathtt{Begin}: & \mathtt{if}\ \mathtt{x} = 0\ \mathtt{then\ goto\ End} \\ & \mathtt{x} := \mathtt{x} \dot{-} 1 \\ & \mathtt{if}\ \mathtt{y} = 0\ \mathtt{then\ goto\ Begin}\end{array}$$

Goto

- `goto mylabel;`

  stands for the (labelled) URM statement

  `if aux0` $= 0$ `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.

$\texttt{while}\ (\texttt{x} \neq 0)\ \texttt{do}\ \{\cdots\}$

$\texttt{while}\ (\texttt{x} \neq 0)\ \texttt{do}\ \{$
$\quad \langle Instructions \rangle \};$

stands for the following URM program:

```
LabelLoop :   if $\texttt{x} = 0$ then goto End;
              $\langle Instructions \rangle$
              goto LabelLoop;
```

Repeat Loop

```
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.

$\mathtt{x} := 0$

$\mathtt{x} := 0$

stands for the following program:

$$\mathtt{while}\ (\mathtt{x} \neq 0)\ \mathtt{do}\ \{\mathtt{x} := \mathtt{x} \dot{-} 1;\};$$

$\mathrm{y} := \mathrm{x};$

$\mathrm{y} := \mathrm{x};$
stands for (if x, y denote different registers, `aux` is new):

```
$\mathtt{aux} := 0$
while $(\mathrm{x} \neq 0)$ do {
    $\mathrm{x} := \mathrm{x} \dot{-} 1;$
    $\mathtt{aux} := \mathtt{aux} + 1;$ };          $--\mathrm{x} = 0; \mathtt{aux} = \mathrm{x} \sim$
$\mathrm{y} := 0;$                                  $--\mathrm{x} = \mathrm{y} = 0; \mathtt{aux} = \mathrm{x} \sim$
while $(\mathtt{aux} \neq 0)$ do {
    $\mathtt{aux} := \mathtt{aux} \dot{-} 1;$
    $\mathrm{x} := \mathrm{x} + 1;$
    $\mathrm{y} := \mathrm{y} + 1;$ };              $--\mathrm{x} = \mathrm{x} \sim; \mathrm{y} = \mathrm{x} \sim; \mathtt{aux} = 0;$
```

If x, y are the same register, $\mathrm{y} := \mathrm{x}$ stands for the empty statement.

Notation x$\sim$

- On the previous slide the comments (indicated by $--$) indicate the state of the variables after executing this statement.
- $\mathtt{x}\sim$, $\mathtt{y}\sim$ denote the values of $\mathtt{x}$, $\mathtt{y}$ before executing the procedure.

Aliasing Problem

- Note that if for $\mathtt{x}$, $\mathtt{y}$ denoting the same register we would define $\mathtt{y} := \mathtt{x}$ as the same program as when they are different (using a while loop) we obtain the following program (comments explain the effects in this case):

```
$\mathtt{aux} := 0$
$\mathtt{while}\ (\mathtt{x} \neq 0)\ \mathtt{do}\ \{$
  $\mathtt{x} := \mathtt{x} \dot{-} 1;$
  $\mathtt{aux} := \mathtt{aux} + 1;\ \};$          $--\mathtt{x} = 0; \mathtt{aux} = \mathtt{x}\sim$
$\mathtt{x} := 0;$                                 $--\mathtt{x} = 0; \mathtt{aux} = \mathtt{x}\sim$
$\mathtt{while}\ (\mathtt{aux} \neq 0)\ \mathtt{do}\ \{$
  $\mathtt{aux} := \mathtt{aux} \dot{-} 1;$
  $\mathtt{x} := \mathtt{x} + 1;$
  $\mathtt{x} := \mathtt{x} + 1;\ \};$             $--\mathtt{x} = \mathtt{x}\sim \cdot 2; \mathtt{aux} = 0;$
```

Aliasing Problem

- Instead of assigning `x` to `y` (which means doing nothing), `x` is doubled in this program.
- The above is an occurrence of the **aliasing problem**.
- The aliasing problem occurs if we have procedure with parameters which modifies its arguments, and if this program doesn't do what it is intended to do in case two of its arguments are instantiated by the same variable.
- Frequent reason for programming erros, which are difficult to detect.

$y := x;$

- 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

$$\begin{aligned} x &:= y + z \\ x &:= y \dot{-} z \end{aligned}$$

the values of y and z will preserved.

$x := y + z;$

Assume x, y, z denote different registers.

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

```
$x := y;$                          $-- x = y \sim; y = y \sim$
$aux := z;$
while $(aux \neq 0)$ do $\{$
    $aux := aux \dot{-} 1;$
    $x := x + 1; \};$              $-- x = y \sim + z \sim;$
                                   $-- y = y \sim; z = z \sim; aux = 0;$
```

$\mathtt{x} := \mathtt{y} \dot{-} \mathtt{z};$

Assume x, y, z denote different registers.
Remember, that $a \dot{-} b := \max\{0, a - b\}$.
$\mathtt{x} := \mathtt{y} \dot{-} \mathtt{z};$
is computed as follows (aux is an additional variable):

```
$\mathtt{x} := \mathtt{y};$
$\mathtt{aux} := \mathtt{z};$
while $(\mathtt{aux} \neq 0)$ do {
    $\mathtt{aux} := \mathtt{aux} \dot{-} 1;$
    $\mathtt{x} := \mathtt{x} \dot{-} 1;$ };
```

Checking for Inequality

- We have

$$(x \dot{-} y) + (y \dot{-} x) \neq 0 \Leftrightarrow x \neq y$$

- **Proof:**
  - If $x > y$, then

$$\begin{aligned} x \dot{-} y &> 0 \ , \\ y \dot{-} x &= 0 \ , \\ (x \dot{-} y) + (y \dot{-} x) &> 0 \end{aligned}$$

  - If $y > x$, then

$$\begin{aligned} y \dot{-} x &> 0 \ , \\ x \dot{-} y &= 0 \ , \\ (x \dot{-} y) + (y \dot{-} x) &> 0 \end{aligned}$$

Checking for Inequality

$$(x \dot{-} y) + (y \dot{-} x) \neq 0 \Leftrightarrow x \neq y$$

- If $x = y$, then

$$\begin{aligned} y \dot{-} x &= 0 \ , \\ x \dot{-} y &= 0 \ , \\ (x \dot{-} y) + (y \dot{-} x) &= 0 \end{aligned}$$

Checking for Inequality

$(\mathrm{x} \dot{-} \mathrm{y}) + (\mathrm{y} \dot{-} \mathrm{x}) \neq 0 \Leftrightarrow \mathrm{x} \neq \mathrm{y}$

- So a while loop

$$\texttt{while}\ (\mathrm{x} \neq \mathrm{y})\ \texttt{do}\ \{\cdots\}$$

can be replaced by

$$\texttt{while}\ ((\mathrm{x} \dot{-} \mathrm{y}) + (\mathrm{y} \dot{-} \mathrm{x}) \neq 0)\ \texttt{do}\ \{\cdots\}$$

Checking for Inequality

$\texttt{while}\ ((\mathrm{x} \dot{-} \mathrm{y}) + (\mathrm{y} \dot{-} \mathrm{x}) \neq 0)\ \texttt{do}\ \{\cdots\}$

which can be replaced by

```
$\texttt{aux} := (\mathrm{x} \dot{-} \mathrm{y}) + (\mathrm{y} \dot{-} \mathrm{x})$
$\texttt{while aux} \neq 0)$ do
$\{\cdots$
  $\texttt{aux} := (\mathrm{x} \dot{-} \mathrm{y}) + (\mathrm{y} \dot{-} \mathrm{x})$
  $\}$
```

If we unfold this further, we obtain the following:

while $(\mathtt{x} \neq \mathtt{y})$ do $\{\cdots\}$

Assume $\mathtt{x}$, $\mathtt{y}$ denote different registers.

```
while $(\mathtt{x} \neq \mathtt{y})$ do $\{$
  $\langle \mathit{Statements} \rangle \}$;
```

stands for ($\mathtt{aux}$, $\mathtt{aux}_i$ denote new registers):

```
$\mathtt{aux}_0 := \mathtt{x} \dot{-} \mathtt{y}$;
$\mathtt{aux}_1 := \mathtt{y} \dot{-} \mathtt{x}$;
$\mathtt{aux} := \mathtt{aux}_0 + \mathtt{aux}_1$;
while $(\mathtt{aux} \neq 0)$ do $\{$
  $\langle \mathit{Statements} \rangle$
  $\mathtt{aux}_0 := \mathtt{x} \dot{-} \mathtt{y}$;
  $\mathtt{aux}_1 := \mathtt{y} \dot{-} \mathtt{x}$;
  $\mathtt{aux} := \mathtt{aux}_0 + \mathtt{aux}_1; \}$;
```

(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.

Notations for Partial Functions

Definition 3.1

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

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

succ$(x) = x + 1$.

(c) Define for $0 \leq i < n$ the **projection function** proj$_i^n : \mathbb{N}^n \rightarrow \mathbb{N}$, 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 zero $: \mathbb{N} \overset{\sim}{\rightarrow} \mathbb{N}$.
- proj$_0^1 : \mathbb{N} \rightarrow \mathbb{N}$ is the identity function: proj$_0^1(x) = x$.

Notations for Partial Functions

(d) Assume

$$\begin{array}{lcl} g & : & (B_0 \times \cdots \times B_{k-1}) \xrightarrow{\sim} C \quad , \\ h_i & : & A_0 \times \cdots \times A_{n-1} \xrightarrow{\sim} B_i \quad . \qquad i = 0, \ldots, k-1 \end{array}$$

Define

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

$$f(\vec{\mathrm{a}}) :\simeq g(h_0(\vec{\mathrm{a}}), \ldots, h_{k-1}(\vec{\mathrm{a}}))$$

- In case of $k = 1$ we write $g \circ h$ instead of $g \circ (h)$.
- Furthermore as usual

$$g_1 \circ g_2 \circ \cdots \circ g_n := g_1 \circ (g_2 \circ (\cdots \circ (g_{n-1} \circ g_n))) \quad .$$

Notations for Partial Functions

(e) Assume

$$\begin{aligned} g &: \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N} \ , \\ h &: \mathbb{N}^{k+2} \overset{\sim}{\rightarrow} \mathbb{N} \ . \end{aligned}$$

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

$$\begin{aligned} f(\vec{n}, 0) &:\simeq g(\vec{n}) \\ f(\vec{n}, m+1) &:\simeq h(\vec{n}, m, f(\vec{n}, m)) \end{aligned}$$

- We write $\mathrm{primrec}(g, h)$ for the function f just defined.
- So $\mathrm{primrec}(g, h) : \mathbb{N}^{k+1} \overset{\sim}{\rightarrow} \mathbb{N}$.

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 $a \in \mathbb{N}$, $h : \mathbb{N}^2 \overset{\sim}{\to} \mathbb{N}$.
Define

$$f : \mathbb{N} \overset{\sim}{\to} \mathbb{N}$$

by primitive recursion from a and h as follows:

$$\begin{array}{rcl} f(0) & :\simeq & a \\ f(m+1) & :\simeq & h(m, f(m)) \end{array}$$

We write $\mathsf{primrec}(a, h)$ for f, so $\mathsf{primrec}(a, h) : \mathbb{N} \overset{\sim}{\to} \mathbb{N}$.

primrec in Haskell

- In Haskell we can define **primrec** as a higher-order function as follows:

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

- - primrec0 is the operator for primitive recursion
- - defining a 1-ary function primrec0 f a :: Nat → Nat
- - from f: Nat → Nat → Nat and a: Nat

primrec0 :: Nat → (Nat → Nat → Nat) → Nat → Nat
primrec0 a g Z = a
primrec0 a g (S n) = g n (primrec0 a g n)
```

primrec in Haskell (Cont.)

```
- - primrec1 is the operator for primitive recursion
- - defining a 2-ary function primrec1 f g :: Nat → Nat → Nat
- - from f: Nat → Nat → Nat → Nat and g: Nat → Nat

primrec1 :: (Nat→ Nat)
            → (Nat → Nat → Nat→ Nat)
            → Nat → Nat → Nat
primrec1 g h n Z = g n
primrec1 g h n (S m) = h n m (primrec1 g h n m)
```

Examples for Primitive Recursion

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

  $$\begin{aligned} \mathsf{add}(x, 0) &= x + 0 = x \\ \mathsf{add}(x, y + 1) &= x + (y + 1) = (x + y) + 1 = \mathsf{add}(x, y) + 1 \end{aligned}$$

  Therefore

  $$\begin{aligned} \mathsf{add}(x, 0) &= g(x) \\ \mathsf{add}(x, y + 1) &= h(x, y, \mathsf{add}(x, y)) \end{aligned}$$

  where

  $$\begin{aligned} g &: \mathbb{N} \to \mathbb{N} \ , & g(x) &:= x \ , \\ h &: \mathbb{N}^3 \to \mathbb{N} \ , & h(x, y, z) &:= z + 1 \ . \end{aligned}$$

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

Addition (add)

$$g : \mathbb{N} \to \mathbb{N}\ , \qquad g(x) := x\ ,$$
$$h : \mathbb{N}^3 \to \mathbb{N}\ , \qquad h(x,y,z) := z+1\ ,$$
$$\mathsf{add} := \mathsf{primrec}(g,h)$$

- We have
  - $\mathsf{add}(x,0) = g(x) = x = x+0$.
  - $\mathsf{add}(x,1) = h(x,0,\mathsf{add}(x,0)) = \mathsf{add}(x,0)+1 = x+1$.
  - $\mathsf{add}(x,2) = h(x,1,\mathsf{add}(x,1)) = \mathsf{add}(x,1)+1 = (x+1)+1$.
  - etc.

Defining $+$ from primrec in Haskell

In Haskell we can define `add` from primrec as follows

$\text{add} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$

$\text{add} = \text{primrec1}\ (\lambda \text{n} \rightarrow \text{n})\ (\lambda \text{n m k} \rightarrow \text{S k})$

Examples for Primitive Recursion

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

$$\begin{aligned} \mathsf{mult}(x, 0) &= x \cdot 0 = 0 \\ \mathsf{mult}(x, y+1) &= x \cdot (y+1) = x \cdot y + x = \mathsf{mult}(x, y) + x \end{aligned}$$

  Therefore

$$\begin{aligned} \mathsf{mult}(x, 0) &= g(x) \\ \mathsf{mult}(x, y+1) &= h(x, y, \mathsf{mult}(x, y)) \end{aligned}$$

  where

$$\begin{aligned} g : \mathbb{N} \to \mathbb{N} \quad &, \quad g(x) := 0 \quad , \\ h : \mathbb{N}^3 \to \mathbb{N} \quad &, \quad h(x, y, z) := z + x \quad . \end{aligned}$$

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

Multiplication (mult)

$$\begin{aligned} g : \mathbb{N} \to \mathbb{N} \ , &\qquad g(x) := 0 \ , \\ h : \mathbb{N}^3 \to \mathbb{N} \ , &\qquad h(x, y, z) := z + x \ , \end{aligned}$$

$$\mathrm{mult} := \mathrm{primrec}(g, h)$$

- We have
  - $\mathrm{mult}(x, 0) = g(x) = 0 = x \cdot 0$.
  - $\mathrm{mult}(x, 1) = h(x, 0, \mathrm{mult}(x, 0)) = \mathrm{mult}(x, 0) + x = 0 + x = x$.
  - $\mathrm{mult}(x, 2) = h(x, 1, \mathrm{mult}(x, 1)) = \mathrm{mult}(x, 1) + x = (x \cdot 1) + x$.
  - etc.

Examples for Primitive Recursion

- Let pred $: \mathbb{N} \to \mathbb{N}$,

$$\mathrm{pred}(n) := n \dot{-} 1 = \begin{cases} n - 1 & \text{if } n > 0, \\ 0 & \text{otherwise.} \end{cases}$$

pred can be defined using primitive recursion:

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(x+1) &= x \end{aligned}$$

Therefore

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(x+1) &= h(x, \mathrm{pred}(x)) \end{aligned}$$

where

$$h : \mathbb{N}^2 \to \mathbb{N} \quad , \qquad h(x, y) := x$$

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

Examples for Primitive Recursion

- $x \dot{-} y$ can be defined using primitive recursion:
  Let $f(x,y) := x \dot{-} y$. We have

$$\begin{aligned} f(x,0) &= x \dot{-} 0 = x \\ f(x,y+1) &= x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 \\ &= \text{pred}(x \dot{-} y) = \text{pred}(f(x,y)) \end{aligned}$$

Therefore

$$\begin{aligned} f(x,0) &= g(x) \\ f(x,y+1) &= h(x,y,f(x,y)) \end{aligned}$$

where

$$\begin{aligned} g : \mathbb{N} \to \mathbb{N} \quad &, \qquad\quad g(x) := x \quad , \\ h : \mathbb{N}^3 \to \mathbb{N} \quad &, \quad h(x,y,z) := \text{pred}(z) \quad . \end{aligned}$$

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

Remark

- If $f = \mathrm{primrec}(g, h)$, then

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

- **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

$$h \;:\; \mathbb{N}^2 \xrightarrow{\sim} \mathbb{N}\ , \qquad h(n,m) \;\simeq\; \begin{cases} m \dot{-} 1 & \text{if } m > 0, \\ \perp & \text{otherwise.} \end{cases}$$

- Let

$$\begin{array}{rll} & f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}\ , & f := \text{primrec}(1, h)\ , \\ \text{i.e.} & f(0) \simeq 1\ , & f(n+1) \simeq h(n, f(n))\ . \end{array}$$

- Then

$$\begin{aligned} f(0) &\simeq 1 \\ f(1) &\simeq h(0, f(0)) \simeq h(0,1) \simeq 0 \\ f(2) &\simeq h(1, f(1)) \simeq h(1,0)\uparrow \\ \forall m \geq 2.&f(m)\uparrow \end{aligned}$$

Primitive-Recursive Functions

- The functions, which can be defined from zero, succ, proj_i^k by using composition ($\circ$) and primitive recursion (primrec) are called the **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 **not Turing-complete**.

Notations for Partial Functions

- Let $g : \mathbb{N}^{n+1} \overset{\sim}{\to} \mathbb{N}$.
  We define $\mu y.(g(\vec{x}, y) \simeq 0)$:

$$\mu y.(g(\vec{x}, y) \simeq 0) :\simeq \begin{cases} \text{the least } y \in \mathbb{N} \text{ s.t.} & \\ \quad g(\vec{x}, y) \simeq 0 & \\ \quad \text{and for } 0 \leq y' < y & \\ \quad \text{there exists a } z' \neq 0 & \\ \quad \text{s.t. } g(\vec{x}, y') \simeq z' & \text{if such } y \\ & \text{exists,} \\ \\ \bot & \text{otherwise} \end{cases}$$

$\mu(g)$

- Now define $h : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$,

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

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

Examples

- Assume

$$\begin{aligned} g(x,0) &\simeq 1 \\ g(x,1) &\uparrow \\ g(x,2) &\simeq 0 \end{aligned}$$

Then

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

- Assume instead

$$\begin{aligned} g(x,0) &\simeq 1 \\ g(x,1) &\simeq 5 \\ g(x,2) &\simeq 0 \end{aligned}$$

Then

$$\mu y.(g(x,y) \simeq 0) \simeq 2$$

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.

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

Computation of $\mu(g)$

- If we defined $\mu(g)(\vec{x})$ to be the least y 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 μ

- Let $f : \mathbb{N}^2 \to \mathbb{N}$, $f(x, y) := x \dot{-} y$. Then
$$\mu y.(f(x, y) \simeq 0) \simeq x$$
so $\mu(f)(x) \simeq x$.

- Let $f : \mathbb{N} \overset{\sim}{\to} \mathbb{N}$,
$f(0)\uparrow$,
$f(n) := 0$ for $n > 0$.
Then
$$\mu y.(f(y) \simeq 0)\uparrow$$
.

Examples for μ

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

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

  $\mu y.(f(y) \simeq 0)$ 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 ≥ 4 is the sum of two primes.

  This is equivalent to $\mu y.(\text{‘}f(y) \simeq 0)\uparrow$.

  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, succ, proj_i^k by using composition ($\circ$) and primitive recursion (primrec)

  but by additionally closing them under μ, 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.

Lemma and Definition 3.2

Assume $f : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ *is URM-computable.*
Assume $\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$, y, z_0, ... ,z_l *are different variables.*
Then one can define a URM program, which, computes $f(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1})$ *and stores the result in* y *in the following sense:*

- *If* $f(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}) \downarrow$, *the program terminates at the first instruction following this* **program**, *and stores the result in* y.
- *If* $f(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1})\uparrow$, *the program never terminates.*

The program can be defined so that it doesn't change $\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}, \mathrm{z}_0, \ldots, \mathrm{z}_l$.

For U *we say it is* **a URM program which computes $\mathrm{y} \simeq f(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1})$ and preserves $\mathrm{z}_0, \ldots, \mathrm{z}_l$**.

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 choosen 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, and set variables $\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$, $\mathrm{z}_0, \ldots, \mathrm{z}_l$ back to their original (stored) values.

Omit Proof.

Proof

Let U be a URM program s.t. $\mathrm{U}^{(k)} = f$.
Let $\mathrm{u}_0, \ldots, \mathrm{u}_{k-1}$ be registers different from the above.
By renumbering of registers and of jump addresses, we obtain a program U', which computes the result of $f(\mathrm{u}_0, \ldots, \mathrm{u}_{k-1})$ 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:

$\mathrm{u}_0 := \mathrm{x}_0;$

$\cdots$

$\mathrm{u}_{k-1} := \mathrm{x}_{k-1};$
U'

$\mathrm{y} := \mathrm{u}_0;$

Lemma 3.3

(a) zero, succ *and* proj_i^n *are URM-computable.*

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

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

(d) If $g : \mathbb{N}^{n+1} \stackrel{\sim}{\to} \mathbb{N}$ *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.

Omit Proof.

Proof of Lemma 3.3 (a)

Let x_i denote register R_i.

Proof of (a)

- zero is computed by the following program:

  $\mathrm{x}_0 := 0$.

- succ is computed by the following program:

  $\mathrm{x}_0 := \mathrm{x}_0 + 1$.

- proj^n_k is computed by the following program:

  $\mathrm{x}_0 := \mathrm{x}_k$.

  - Especially, if $k = 0$ then proj^n_k is the empty program (i.e. the program with no instructions this is since we defined $\mathrm{x}_0 := \mathrm{x}_0$ to be the empty program.)

Proof of Lemma 3.3 (b)

Assume $f : \mathbb{N}^n \overset{\sim}{\rightarrow} \mathbb{N}$, $g_i : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N}$ 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 $\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$.
  Let $\vec{\mathrm{x}} := \mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$.
- First we compute $g_i(\vec{\mathrm{x}})$ 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 $\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$ and the previously computed values $g_i(\vec{\mathrm{x}})$, which are stored in y_j for $j < i$ are not destroyed.
- Then compute $f(\mathrm{y}_0, \ldots, \mathrm{y}_{n-1})$, and store result in x_0.
- Then x_0 contains $f(g_0(\vec{\mathrm{x}}), \ldots, g_{n-1}(\vec{\mathrm{x}}))$.

Proof of Lemma 3.3 (b)

- Let therefore U_i be a URM program $(i = 0, \ldots, n-1)$, which computes $\mathrm{y}_i \simeq g_i(\vec{\mathrm{x}})$ and preserves y_j for $j \neq i$.
- Let V be a URM program, which computes $\mathrm{x}_0 \simeq f(\mathrm{y}_0, \ldots, \mathrm{y}_{n-1})$.

Proof of Lemma 3.3 (b)

Let U' be defined as follows:

U_0

$\cdots$

U_{n-1}

V

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

Omit rest of proof.

Proof of Lemma 3.3 (b)

U' is the program

U_0
$\cdots$
U_{n-1}
V

- **Case 1:** For one i $g_i(\vec{\mathrm{x}})\uparrow$.
  The program will loop in program U_i for the first such i.
  $\mathrm{U}'^{(k)}(\vec{\mathrm{x}})\uparrow$, $f \circ (g_0, \ldots, g_{n-1})(\vec{\mathrm{x}})\uparrow$.
- **Case 2:** For all i $g_i(\vec{\mathrm{x}}) \downarrow$.
  The program executes U_i, sets $\mathrm{y}_i \simeq g_i(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1})$ and reaches beginning of V.

Proof of Lemma 3.3 (b)

U' is the program

U_0
$\cdots$
U_{n-1}
V

- **Case 2.1:** $f(g_0(\vec{\mathrm{x}}), \ldots, g_{n-1}(\vec{\mathrm{x}}))\uparrow$.
  V will loop, $\mathrm{U}'^{(k)}(\vec{\mathrm{x}})\uparrow$, $f \circ (g_0, \ldots, g_{n-1})(\vec{\mathrm{x}})\uparrow$.
- **Case 2.2:** Otherwise.
  The program reaches the end of program V and result in $\mathrm{x}_0 \simeq f(g_0(\vec{\mathrm{x}}), \ldots, g_{n-1}(\vec{\mathrm{x}}))$.
  So $\mathrm{U}'^{(k)}(\vec{\mathrm{x}}) \simeq (f \circ (g_0, \ldots, g_{n-1}))(\vec{\mathrm{x}})$.

Proof of Lemma 3.3 (b)

In all cases

$$\mathrm{U}'^{(k)}(\vec{\mathrm{x}}) \simeq (f \circ (g_0, \ldots, g_{n-1}))(\vec{\mathrm{x}}) \quad .$$

Proof of Lemma 3.3 (c)

Assume

$$g : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N} \ , \quad h : \mathbb{N}^{n+2} \overset{\sim}{\to} \mathbb{N}$$

are URM-computable.
Let

$$f := \text{primrec}(g, h) \ .$$

Show f is URM-computable.
Defining equations for f are as follows (let $\vec{n} := n_0, \ldots, n_{n-1}$):

- $f(\vec{n}, 0) \simeq g(\vec{n})$,
- $f(\vec{n}, k + 1) \simeq h(\vec{n}, k, f(\vec{n}, k))$.

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.
- $\cdots$
- 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 $\vec{\mathrm{x}} := \mathrm{x}_0, \ldots, \mathrm{x}_{n-1}$.
  Let y, z, u be new registers.
- Compute $f(\vec{\mathrm{x}}, \mathrm{y})$ for $\mathrm{y} = 0, 1, 2, \ldots, \mathrm{x}_n$, and store result in z.
  - Initially we have $\mathrm{y} = 0$ (holds for all registers except of $\mathrm{x}_0, \ldots, \mathrm{x}_n$ initially).
    We compute $\mathrm{z} \simeq g(\vec{\mathrm{x}})$ ($\simeq f(\vec{\mathrm{x}}, 0)$).
    Then $\mathrm{y} = 0$, $\mathrm{z} \simeq f(\vec{\mathrm{x}}, 0)$.

Proof of Lemma 3.3 (c)

- In step from y to $\mathrm{y}+1$:
  - Assume that we have $\mathrm{z} \simeq f(\vec{\mathrm{x}}, \mathrm{y})$.
  - We want that after increasing y by 1 the **loop invariant** $\mathrm{z} \simeq f(\vec{\mathrm{x}}, \mathrm{y})$ still holds.
    Obtained as follows
    - Compute $\mathrm{u} \simeq h(\vec{\mathrm{x}}, \mathrm{y}, \mathrm{z})$ $(\simeq h(\vec{\mathrm{x}}, \mathrm{y}, f(\vec{\mathrm{x}}, \mathrm{y})) \simeq f(\vec{\mathrm{x}}, \mathrm{y}+1))$.
    - Execute $\mathrm{z} := \mathrm{u}$ $(\simeq f(\vec{\mathrm{x}}, \mathrm{y}+1))$.
    - Execute $\mathrm{y} := \mathrm{y}+1$.
    - At the end , $\mathrm{z} \simeq f(\vec{\mathrm{x}}, \mathrm{y})$ for the new value of y.
- Repeat this until $\mathrm{y} = \mathrm{x}_n$.
- Once y has reached x_n, z contains $f(\vec{\mathrm{x}}, \mathrm{y}) \simeq f(\vec{\mathrm{x}}, x_n)$.
- Execute $\mathrm{x}_0 := \mathrm{z}$.

Proof of Lemma 3.3 (c)

Let

- U be a URM program, which computes $\mathrm{z} \simeq g(\vec{\mathrm{x}})$ and preserves y (by definition 3.2, it doesn't modify the arguments $\vec{\mathrm{x}}$ of g);
- V be a program, which computes $\mathrm{u} \simeq h(\vec{\mathrm{x}}, \mathrm{y}, \mathrm{z})$. (by definition 3.2, it doesn't change $\vec{\mathrm{x}}, \mathrm{y}, \mathrm{z}$.)

Proof of Lemma 3.3 (c)

Let U' be as follows:

```
U                      % Compute $\mathtt{z} \simeq g(\vec{\mathrm{x}})(\simeq f(\vec{x}, 0))$
while $(\mathrm{x}_n \neq \mathrm{y})$ do {
    V                  % Compute $\mathtt{u} \simeq h(\vec{\mathrm{x}}, \mathrm{y}, \mathtt{z})$
                       % will be $\simeq h(\vec{\mathrm{x}}, \mathrm{y}, f(\vec{\mathrm{x}}, \mathrm{y})) \simeq f(\vec{\mathrm{x}}, \mathrm{y} + 1)$

    $\mathtt{z} := \mathtt{u};$
    $\mathrm{y} := \mathrm{y} + 1;$ };
$\mathrm{x}_0 := \mathtt{z};$
```

Proof of Lemma 3.3 (c)

Correctness of this program:

- When U has terminated, we have $\mathrm{y} = 0$ and $\mathrm{z} \simeq g(\vec{\mathrm{x}}) \simeq f(\vec{\mathrm{x}}, \mathrm{y})$.
- After each iteration of the while loop, we have $\mathrm{y} := \mathrm{y}' + 1$ and $\mathrm{z} \simeq h(\vec{\mathrm{x}}, \mathrm{y}', \mathrm{z}')$.
  (y', z' are the previous values of y, z, respectively.)
- Therefore we have $\mathrm{z} \simeq f(\vec{\mathrm{x}}, \mathrm{y})$.
- The loop terminates, when y has reached x_n.
  Then z contains $f(\vec{\mathrm{x}}, \mathrm{y})$.
  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, $\mathrm{U}'^{(n+1)}(\vec{\mathrm{x}}, \mathrm{x}_n)\uparrow$.
  - $f(\vec{\mathrm{x}}, k)\uparrow$ for some $k < \mathrm{x}_n$,
  - subsequently $f(\vec{\mathrm{x}}, l)\uparrow$ for all $l > k$.
  - Especially, $f(\vec{\mathrm{x}}, \mathrm{x}_n)\uparrow$.
  - Therefore $f(\vec{\mathrm{x}}, \mathrm{x}_n) \simeq \mathrm{U}'^{(n+1)}(\vec{\mathrm{x}}, \mathrm{x}_n)$.

Proof of Lemma 3.3 (d)

Assume

$$g : \mathbb{N}^{n+1} \overset{\sim}{\rightarrow} \mathbb{N}$$

is URM-computable.
Show

$$\mu(g)$$

is URM-computable as well.
Note $\mu(g)(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1})$ is the minimal z s.t.

$$g(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}, \mathrm{z}) \simeq 0 \quad .$$

Let $\vec{\mathrm{x}} := \mathrm{x}_0, \ldots, \mathrm{x}_{k-1}$ and let y, z be registers different from $\vec{\mathrm{x}}$.

Proof of Lemma 3.3 (d)

Plan for the program:

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

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

and successively increasing y by 1 until we have $\mathrm{z} = 0$.

Proof of Lemma 3.3 (d)

Let U compute

$$\mathtt{z} \simeq g(\mathtt{x}_0, \ldots, \mathtt{x}_{k-1}, \mathtt{y}) \quad ,$$

(and preserve the arguments $\mathtt{x}_0, \ldots, \mathtt{x}_{k-1}, \mathtt{y}$.)
Let V be as follows:

```
repeat{
  U
  $\mathtt{y} := \mathtt{y} + 1;$ }
until $(\mathtt{z} = 0);$
$\mathtt{y} := \mathtt{y} \dot{-} 1;$
$\mathtt{x}_0 := \mathtt{y};$
```

Omit rest of proof.

Proof of Lemma 3.3 (d)

V is $\quad \texttt{repeat}\{\mathrm{U}; \mathrm{y} := \mathrm{y} + 1;\} \ \texttt{until}\ (\mathrm{z} = 0);$

$\quad \mathrm{y} := \mathrm{y} \dot{-} 1; \mathrm{x}_0 := \mathrm{y};$

Initially $\mathrm{y} = 0$.
After each iteration of the repeat loop, we have

$$\mathrm{y} := \mathrm{y}' + 1 \ \ , \mathrm{z} \simeq g(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}, \mathrm{y}')$$

(y' is the value of y before this iteration).
If the loop terminates, we have

$$\mathrm{z} \simeq 0 \qquad \mathrm{y} = \mathrm{y}' + 1$$

where y' is the first value, such that $g(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}, \mathrm{y}') \simeq 0$.

Proof of Lemma 3.3 (d)

- Finally y is decreased by one.
- Then y is the least y s.t.

$$g(\mathrm{x}_0, \ldots, \mathrm{x}_{k-1}, \mathrm{y}) \simeq 0 \quad .$$

- x_0 is then set to that value.