IV.2 (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.
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.

In this module we will discuss 2 models of computation:
- The **URM**.
  - **Minimalised** version of a **machine language** of a computer.
  - Model which represents what can be carried out on a computer with a **von Neumann architecture**.
- The **Turing machine**.
  - Abstraction of **computation on a piece of paper**.

There are other models of computation.
For instance the set of functions computable by a **Java program** forms a Turing complete model of computation.
IV.2 (a) Definition 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 $I_0, I_1, I_2, \ldots, I_n$;
- 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.

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.
IV.2 (a) Definition of the URM

The URM

\[ R_0 R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 R_9 \ldots \]

\[ I_0 I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 I_9 \]

Program has terminated

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
    \[ R_k := R_k - 1 \]
  
- Here
  \[ x \div y := \max\{x - y, 0\} \],
  i.e.
  \[ x \div y = \begin{cases} x - y & \text{if } y \leq x, \\ 0 & \text{otherwise.} \end{cases} \]
IV.2 (a) Definition of the URM

URM Instructions

- The **conditional jump instruction**

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

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

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

Finiteness

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

Example of a URM Program

- The following is an example of a URM-program:

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

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
IV.2 (a) Definition of the URM

Operation 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 \( R_0 := 0 \).

IV.2 (a) Definition of the URM

URM-Computable Functions

- For every URM-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 \( k \geq 1 \).

Partial Functions

- The functions \( U^{(1)} \), \( U^{(2)} \), \ldots will be partial, since not for all inputs we obtain an output.
- A partial function \( f : A \rightarrow B \) is a function mapping some elements of \( A \) to elements of \( B \).
- We write
  - \( f(a) \downarrow \) for "\( f(a) \) is defined" (\( f(a) \) returns an element of \( B \)).
  - \( f(a) \uparrow \) for "\( f(a) \) is undefined".
  - \( f(a) \simeq t \) ("\( f(a) \) is partially equal to term \( t \)"") for "\( f(a) \) and \( t \) are both undefined or both defined and return the same value".
  - \( f(a) = t \) for "both \( f(a) \) and \( t \) are defined and return the same value".
  - \( \bot \) for the term which is always undefined (pronounced "bottom").

Partial Functions

- So in case \( f(a) \simeq g(a') \) we only demand that if one of \( f(a) \) or \( g(a') \) are defined then both are defined and return the same result.
- If we write \( f(a) = g(a') \) we demand that both \( f(a) \) and \( g(a') \) are defined and return the same value.
- \( f(a) \simeq \bot \) means the same as \( f(a) \uparrow \).
  - Both are equivalent to "\( f(a) \) is undefined".
- \( f(a) \simeq 3 \) means the same as \( f(a) = 3 \)
  - Since \( 3 \) is defined, \( f(a) \simeq 3 \) implies \( f(a) \downarrow \), and therefore both \( f(a) \simeq 3 \) and \( f(a) = 3 \) are equivalent to "\( f(a) \) is defined and its value is equal to \( 3 \)".
IV.2 (a) Definition of the URM

Domain Theory

▶ There is a theory called “domain theory” in which there is an ordering on the definedness of objects.
▶ For instance if \( f, g : \mathbb{N} \rightarrow \mathbb{N} \) only differ by \( f(0) \downarrow, g(0) \uparrow \), then we can consider \( g \) to be more defined then \( f \).
▶ \( \perp \) is the completely undefined element, therefore it is called **bottom** for being the least element in this order.

**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).

**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
\[
U^{(k)}(a_0, \ldots, a_{k-1}) \simeq b .
\]
  ▶ If the program never stops, 
\[
U^{(k)}(a_0, \ldots, a_{k-1}) \uparrow .
\]
▶ \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) is **URM-computable**, if \( f = U^{(k)} \) for some \( k \in \mathbb{N} \) and some URM program \( U \).
Consider the example of a URM-program treated before:

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

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

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

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 \( U^{(2)}(k, 0) \approx 0 \).

What is \( U^{(2)}(k, l) \) for \( l > 0 \)?

For a \textbf{partial} function \( f \) to be computable we need only:

\begin{itemize}
  \item If \( f(a) \downarrow \), then after finite amount of time we can determine this property, and the value of \( f(a) \).
  \item If \( f(a) \uparrow \), we will wait infinitely long for an answer, so we never determine that \( f(a) \uparrow \).
\end{itemize}

\textbf{Turing halting problem} is the question: “Is \( f(a) \downarrow \)?”.

Turing halting problem is \textbf{undecidable}.

If we want to have always an answer, we need to refer to \textbf{total computable functions}. 
IV.2 (a) Definition of the URM

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:

\[
R_0 := R_0 + R_1
\]

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]

- 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 \( R_1 \).

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]

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

\[
\text{while } (R_1 \neq 0) \text{ do } \{ R_0 := R_0 + 1 \\
R_1 := R_1 - 1 \}
\]
Example of URM-Comp. Function

LabelBegin:  if $R_1 = 0$ then goto LabelEnd;
            $R_0 := R_0 + 1$; $R_1 := R_1 - 1$; 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;
            $R_0 := R_0 + 1$;
            $R_1 := R_1 - 1$;
            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: $R_1 := R_1 - 1$;
3: if $R_2 = 0$ then goto 0;
4: 

Step 6:
Translate the program into a URM program $I_0, I_1, I_2, I_3$:
$I_0 = \text{ifzero}(1, 4)$
$I_1 = \text{succ}(0)$
$I_2 = \text{pred}(1)$
$I_3 = \text{ifzero}(2, 0)$
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 \( \text{succ}(0) \)
  \[
  \begin{align*}
  \text{U} \\
  \text{pred}(0)
  \end{align*}
  \]
  we add 1 to the jump addresses in the original version of \( U \).
- Furthermore, we assume that, if \( U \) 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
  - \( R_k := R_k + 1 \) for \( \text{succ}(k) \),
  - \( R_k := R_k - 1 \) for \( \text{pred}(k) \),
  - if \( R_k = 0 \) then goto \( q \) for \( \text{ifzero}(k, q) \).

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

End:

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

Replacing Registers by Variables

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

```
Begin: if R_0 = 0 then goto End
R_0 := R_0 - 1
if R_1 = 0 then goto Begin
```

the following

```
Begin: if x = 0 then goto End
x := x - 1
if y = 0 then goto Begin
```

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.

Goto

So

```
LabelLoop: if x = 0 then goto End;
x := x - 1
goto LabelLoop;
```

stands for

```
LabelLoop: if x = 0 then goto End;
x := x - 1
if aux0 = 0 then goto LabelLoop;
```

for a new register aux0.
IV.2 (b) Higher level programming concepts for URMs

Repeat Loop

A repeat loop has the form:

```
repeat
  ⟨Instructions⟩
until ⟨condition⟩;
```

A repeat loop is executed by running the body again and again, until at the end of running it until ⟨condition⟩ is true.

So the loop is executed at least one, and then executed iteratively as long as ⟨condition⟩ is false.

So it is equivalent to

```
⟨Instructions⟩
while ¬⟨condition⟩ do {
  ⟨Instructions⟩
}
```

So a repeat loop

```
repeat{
  ⟨Instructions⟩
until x = 0;
```

can be replaced by 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.

while (x ≠ 0) do {⋯}
y := x;

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

```
aux := 0
while (x ≠ 0) do {
    x := x − 1;
    aux := aux + 1;
} ;
−−x = 0; aux = x ∼
y := 0;
```

```
while (aux ≠ 0) do {
    aux := aux − 1;
    x := x + 1;
    y := y + 1;
} ;
−−x = x ∼ ; y = x ∼ ; aux = 0;
```

On the previous slide the comments (indicated by −−) indicate the state of the variables after executing this statement.

x ∼, y ∼ denote the values of x, y before executing the procedure.

So aux = x ∼ means that aux has now the value of x as it was at the beginning of this piece of code.

Aliasing Problem and y := x

- If x, y are the same register, the previous program doesn't work.
- The above program would look in this case as follows:

```
aux := 0
while (x ≠ 0) do {
    x := x − 1;
    aux := aux + 1;
} ;
−−x = 0; aux = x ∼
x := 0;
```

```
while (aux ≠ 0) do {
    aux := aux − 1;
    x := x + 1;
    y := y + 1;
} ;
−−x = x ∼ ; y = x ∼ ; aux = 0;
```

Aliasing Problem

- Instead of assigning x to y (which means doing nothing), x is doubled in this program.
- So we need to make a special definition in case x and y denote the same register:
  - If x and y denote the same register, then y := x denotes the empty program (no instruction).
- 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 errors, which are difficult to detect.
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
\[
\begin{align*}
x &:= y + z \\
x &:= y - z
\end{align*}
\]
the values of \( y \) and \( z \) will preserved.

Assume \( x, y, z \) denote different registers.
\( x := y + z \); stands for the following program (aux is an additional variable):
\[
\begin{align*}
x &:= y; \quad \text{-- } x = y \sim; y = y \sim \aux &:= z; \\
\text{while } (\aux \neq 0) \text{ do } \\
\quad &\aux := \aux - 1; \\
x &:= x + 1; \quad \text{-- } x = y \sim + z \sim; \\
\quad &y = y \sim; z = z \sim; \aux = 0;
\end{align*}
\]

Checking for Inequality

Assume \( x, y, z \) denote different registers. Remember, that \( a - b := \max\{0, a - b\} \).
\[
x := y - z;
\]
is computed as follows (aux is an additional variable):
\[
\begin{align*}
x &:= y; \\
\aux &:= z; \\
\text{while } (\aux \neq 0) \text{ do } \\
\quad &\aux := \aux - 1; \\
x &:= x - 1;
\end{align*}
\]

We have
\[
(x - y) + (y - x) \neq 0 \iff x \neq y
\]

Proof:
\[
\begin{align*}
\text{-- } x > y, & \quad x - y > 0, \\
& \quad y - x = 0, \\
& \quad (x - y) + (y - x) > 0 \\
\text{-- } y > x, & \quad y - x > 0, \\
& \quad x - y = 0, \\
& \quad (x - y) + (y - x) > 0
\end{align*}
\]
Checking for Inequality

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

- If \(x = y\), then

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

So a while loop

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

can be replaced by

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

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

\[
\text{while } (x \neq y) \text{ do } \\
\langle \text{Statements} \rangle;
\]

stands for (aux, aux; denote new registers):

\[
\begin{align*}
\text{aux}_0 & := x \div y; \\
\text{aux}_1 & := y \div x; \\
\text{aux} & := \text{aux}_0 + \text{aux}_1; \\
\text{while } (\text{aux} \neq 0) \text{ do } \\
\langle \text{Statements} \rangle \\
\text{aux}_0 & := x \div y; \\
\text{aux}_1 & := y \div x; \\
\text{aux} & := \text{aux}_0 + \text{aux}_1;
\end{align*}
\]
IV.2 (a) Definition of the URM

IV.2 (b) Higher level programming concepts for URMs

IV.2 (c) URM computable functions

All material in this section has been moved to “Additional Material”.