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.

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.

- 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.
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.

The URM

\[
\begin{array}{cccccccccc}
R_0 & R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 & R_8 & R_9 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
I_0 & I_1 & I_2 & I_3 & I_4 & I_5 & I_6 & I_7 & I_8 & I_9 \\
\end{array}
\]

PC

Execute Instruction
IV.2 (a) Definition of the URM

The URM

\[
\begin{array}{cccccccccc}
R_0 & R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 & R_8 & R_9 & \cdots \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
I_0 & I_1 & I_2 & I_3 & I_4 & I_5 & I_6 & I_7 & I_8 & I_9 \\
\end{array}
\]

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.
    
    \( \rightarrow \) execute next instruction or terminate.
  
  - A more readable notation is
    
    \[
    R_k := R_k + 1
    \]

- The **predecessor instruction**
  
  \[
  \text{pred}(k),
  \]
  
  where \( k \in \mathbb{N} \).
  
  - Execution:
    
    If \( R_k \) contains value \( > 0 \), decrease the content by 1.
    
    If \( R_k \) contains value 0, leave it as it is.
    
    In all cases increment PC by 1.
  
  - A more readable notation is
    
    \[
    R_k := R_k - 1
    \]

Here

\[
\begin{align*}
x \div y & := \max\{x - y, 0\},
\end{align*}
\]

i.e.

\[
\begin{align*}
x \div y & = \begin{cases} 
    x - y & \text{if } y \leq x, \\
    0 & \text{otherwise.}
\end{cases}
\end{align*}
\]
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:

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

Example

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

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 \( R_0 := 0 \).

Partial Functions

- The functions \( U^{(1)}, U^{(2)}, \ldots \) will be partial, since not for all inputs we obtain an output.
- A partial function \( f : A \sim B \) 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".
  - \( f(a) \sim t \) ("\( f(a) \) 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").
Domain Theory

- There is a theory called “domain theory” in which there is an ordering on the definedness of objects.
- For instance if \( f, g : \mathbb{N} \sim \mathbb{N} \) only differ by \( f(0) \downarrow, g(0) \uparrow \), then we can consider \( g \) to be more defined than \( f \).
- \( \bot \) 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 
  \[
  U^{(k)} : \mathbb{N}^k \sim \mathbb{N}
  \]
  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,
    \[
    U^{(k)}(a_0, \ldots, a_{k-1}) \uparrow .
    \]

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

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 \( U^{(1)}(k) \simeq 0 \).

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 \( l > 0 \)?

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

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

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

- If we want to have always an answer, we need to refer to total computable functions.
In order to describe the total computable functions, we need to introduce the partial computable functions first.

- There is no program language such that 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:

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

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:

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

- 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:

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

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

\[
\text{LabelEnd : }
\]

\[
R_0 := R_0 + R_1
\]
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**: 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.

**Step 4**: Replace last goto by a conditional goto, depending on \( R_2 = 0 \).

**Step 5**: Resolve labels:

<table>
<thead>
<tr>
<th>Label</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if ( R_1 = 0 ) then goto 4;</td>
</tr>
<tr>
<td>1</td>
<td>( R_0 := R_0 + 1; )</td>
</tr>
<tr>
<td>2</td>
<td>( R_1 := R_1 - 1; )</td>
</tr>
<tr>
<td>3</td>
<td>if ( R_2 = 0 ) then goto 0;</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Step 6**: Translate the program into a URM program \( I_0, I_1, I_2, I_3 \):  
\[ I_0 = \text{ifzero}(1, 4) \]
\[ I_1 = \text{succ}(0) \]
\[ I_2 = \text{pred}(1) \]
\[ I_3 = \text{ifzero}(2, 0) \]
IV.2 (b) Higher level programming concepts for URMs

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 $\text{succ}(0)$
  
  $U$
  
  $\text{pred}(0)$

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

- $R_k := R_k + 1$ 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 $\text{End}$ denotes the first instruction following a program.
- So instead of

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

we write

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

End:
Omitting \( I_k = \)

- We omit now “\( I_k = \)”. 
- Furthermore, labels don’t have to start with \textit{Label}, so we can write \textit{Begin} instead of \textit{LabelBegin}. 
- We obtain the following program:
  
  \begin{verbatim}
  Begin: if \( R_0 = 0 \) then goto End
  \( R_0 := R_0 - 1 \)
  if \( R_1 = 0 \) then goto Begin
  End:
  \end{verbatim}

- Since \textit{End :} is always the first instruction following the program, we will omit the last line \textit{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{verbatim}
Begin: if \( R_0 = 0 \) then goto End
\( R_0 := R_0 - 1 \)
if \( R_1 = 0 \) then goto Begin
\end{verbatim}

the following

\begin{verbatim}
Begin: if \( x = 0 \) then goto End
\( x := x - 1 \)
if \( y = 0 \) then goto Begin
\end{verbatim}

Goto

\begin{verbatim}
goto mylabel;
\end{verbatim}

stands for the (labelled) URM statement

\begin{verbatim}
if aux0 = 0 then goto mylabel;
\end{verbatim}

- 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

\begin{verbatim}
LabelLoop: if \( x = 0 \) then goto End;
   \( x := x - 1 \)
goto LabelLoop;
\end{verbatim}

stands for

\begin{verbatim}
LabelLoop: if \( x = 0 \) then goto End;
   \( x := x - 1 \)
if aux0 = 0 then goto LabelLoop;
\end{verbatim}

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

while \((x \neq 0)\) do \{\ldots\}\n
A repeat loop has the form:

\[
\begin{align*}
\text{repeat} & \{ \\
& \langle \text{Instructions} \rangle \\
\text{until} & \langle \text{condition} \rangle \\
\end{align*}
\]

- A repeat loop is executed by running the body again and again, until at the end of running it until \(\langle \text{condition} \rangle\) is true.
- So the loop is executed at least one, and then executed iteratively as long as \(\langle \text{condition} \rangle\) is false.
- So it is equivalent to

\[
\langle \text{Instructions} \rangle \\
\text{while} \neg \langle \text{condition} \rangle \text{ do } \{ \\
& \langle \text{Instructions} \rangle \\
\}
\]

\[
\text{ Repeat Loop}
\]

So a repeat loop

\[
\text{repeat} \{ \\
& \langle \text{Instructions} \rangle \\
\text{until} & x = 0; \\
\langle \text{Instructions} \rangle \\
\text{while} & (x \neq 0) \text{ do } \{ \\
& \langle \text{Instructions} \rangle \\
\}
\]

- Note that this results in doubling of \(\langle \text{Instructions} \rangle\).
- 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.

CS,236 Sect. IV.2 (b) 49/117

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

\[
x := 0
\]

CS,236 Sect. IV.2 (b) 50/117

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

\[
x := 0
\]

CS,236 Sect. IV.2 (b) 51/117

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

\[
x := 0
\]

CS,236 Sect. IV.2 (b) 52/117
IV.2 (b) Higher level programming concepts for URMs

Stand 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;
−−x = y = 0; aux = x ∼
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

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;
−−x = y = 0; aux = x ∼
while (aux ≠ 0) do {
aux := aux − 1;
x := x + 1;
x := x + 1;};  —−x = x ∼ · 2; aux = 0;
```

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

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.

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

Proof:
- If \( x > y \), then

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

- If \( y > x \), then

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

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

- If \(x = y\), then

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

\[\text{So a while loop} \quad \text{while} (x \neq y) \quad \text{do} \{\ldots\}\]

\[\text{can be replaced by} \quad \text{while} \ ((x - y) + (y - x) \neq 0) \quad \text{do} \{\ldots\}\]

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

\[\begin{align*}
\text{while} ((x - y) + (y - x) \neq 0) \quad \text{do} \{\ldots\}
\end{align*}\]

which can be replaced by

\[\begin{align*}
\text{aux} &:= (x - y) + (y - x) \\
\text{while aux} \neq 0 \quad \text{do} \\
\{\ldots\}
\end{align*}\]

\[\text{aux} := (x - y) + (y - x)\]

If we unfold this further, we obtain the following:

\[\begin{align*}
\text{aux} &:= (x - y) + (y - x) \\
\text{while aux} \neq 0 \quad \text{do} \\
\{\ldots\}
\end{align*}\]

\[\text{aux} := (x - y) + (y - x)\]

\[\text{aux} := \text{aux} + \text{aux};\]

\[\text{while} \ (\text{aux} \neq 0) \quad \text{do} \{\ldots}\]

\[\text{aux} := \text{aux} + \text{aux};\]

\[\text{while} \ (\text{aux} \neq 0) \quad \text{do} \{\ldots\}\]
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”. 