CS_275 Automata and Formal Language Theory

Course Notes
Part III: Limits of Computation
Chapt. III.2: The URM

Anton Setzer

http://www.cs.swan.ac.uk/∼csetzer/lectures/automataFormalLanguage/current/index.html

April 3, 2017

III.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.

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.

Models of Computations Discussed

In this module we will discuss 2 models of computation:

- The **URM**.
  - **Minimised** 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.

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.

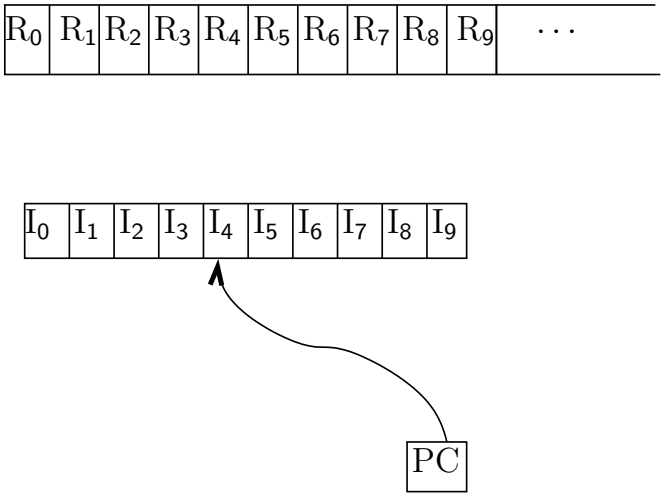
Description of the URM

- 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** $\mathrm{I}_0, \mathrm{I}_1, \mathrm{I}_2, \ldots \mathrm{I}_n$;
  - 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

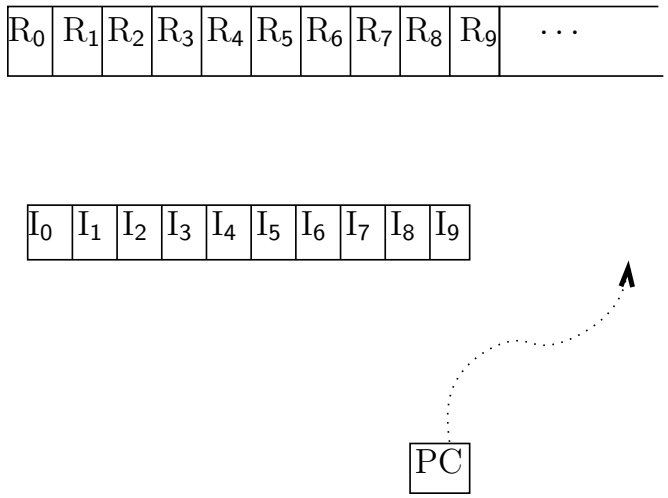
- Note that the URM program is part of the URM.
- One could distinguish between
  - The architecture of a URM consisting of registers, the program counter and a memory for a URM program,
  - and the URM program itself.
- For historic reasons by a URM we mean the URM architecture **together** with a URM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

The URM



Execute Instruction

The URM



Program has terminated

Variants of URM Instructions

- The URM is essentially an abstract version of an **assembly language**.
- As there are many different assembly languages, there are as well **different choices of URM instructions**.
- The concrete choice of URM instructions doesn't matter, as long as they are complete, which means they are **complete**, which means that any other set of URM instructions, which have a computational meaning, can be interpreted into these sets of instructions.
- In the following we are discuss the original set of URM instructions, as introduced by **John Shepherdson**.
- What is interesting that the instructions are **atomic ones**, e.g. they are very simple ones, while still being able to interpret more complex one.

Variants of URM Instructions

- We will then show how to interpret **more complex instructions** such as

$$R_i := R_j$$

into a language having those more atomic instructions.
- Note that we are not looking at programs which are executed **efficiently**.
- Actually programs developed using these instructions will be very **inefficient**.
- All we need is that all computable programs can be interpreted in the URM with the chosen instructions, so that we **capture the notion of a computable function**.

URM Instructions

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

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

where $k \in \mathbb{N}$.

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

  - Until 2012 this instruction was called

succ(k)

URM Instructions

- The **predecessor instruction**

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

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.
- Until 2012 this instruction was called

pred(k)

$x \dot{-} y$

- Here

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

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**

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

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.
- Until 2012 this instruction was called

ifzero(k, 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:

$$\begin{aligned} \mathrm{I}_0 = &\quad \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \\ \mathrm{I}_1 = &\quad \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ \mathrm{I}_2 = &\quad \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0 \end{aligned}$$

- We will write it in more readable form as follows:

$$\begin{aligned} 0: &\quad \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \\ 1: &\quad \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ 2: &\quad \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0 \end{aligned}$$

Example

$0 : \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \qquad 1 : \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1$
$\quad 2 : \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0$

If we run this program with initial 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
0	2	0
1	2	0
2	1	0
0	1	0
1	1	0
2	0	0
0	0	0
3	0	0
URM Stops		

Operation of the Example

$0: \quad \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3$
$1: \quad \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1$
$2: \quad \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0$

- Assume R_1 is initially zero.
- Then R_1 will never be changed by the program, so it will remain 0 for ever.
- So in instruction 2 the URM will always jump to instr. 0.
- Then the program will as long as $\mathrm{R}_0 \neq 0$ 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 $\mathrm{R}_0 := 0$.

URM-Computable Functions

- For every URM-program U we define the function defined by it.
- In fact there are many function which are defined by the same URM-program U:
  - A unary function $\mathrm{U}^{(1)}$ (i.e. a function taking one argument) 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 $\mathrm{U}^{(2)}$ (i.e. a function taking two arguments), which stores its two arguments in R_0 and R_1, then continues operating as $\mathrm{U}^{(1)}$ before.
  - And so on. In general we obtain a k-ary partial function $\mathrm{U}^{(k)}$ for every $k \geq 1$.

Partial Functions

- The functions $\mathrm{U}^{(1)}$, $\mathrm{U}^{(2)}, \ldots$ will be partial, since not for all inputs we obtain an output.
  - If the program runs forever without stopping then we have no output and the result of the function is undefined.
- A partial function $f : A \xrightarrow{\sim} B$ 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".
  - $f(a) \simeq t$ ("$f(a)$ **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".
  - $\perp$ for the term which is always undefined (pronounced "**bottom**").
- A total function $f : A \xrightarrow{\sim} B$ is a partial function, such that for all $a \in A$ we have $f(a)\downarrow$.

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.
- $f(a) \simeq \perp$ 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".

Examples

- Assume U is a URM program.
  - If U started with R_0 containing 3, other registers containing 0 doesn't terminate then

$$\mathrm{U}^{(1)}(3)\uparrow$$
$$\mathrm{U}^{(1)}(3) \simeq \perp$$

  - If U started with R_0 containing 5, other registers containing 0 terminates with R_0 containing 7 then

$$\mathrm{U}^{(1)}(5)\downarrow$$
$$\mathrm{U}^{(1)}(5) \simeq 7$$

Domain Theory

- There is a theory called "**domain theory**" in which there is an ordering on the definedness of objects.
- For instance if $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$ 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 $\mathrm{U}^{(k)}$

- 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 $\mathrm{U}^{(k)}$ taking k natural numbers and returning partially one natural number, i.e.

$$\mathrm{U}^{(k)} : \mathbb{N}^k \rightharpoonup \mathbb{N}$$

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 $\geq n$ (i.e. points to no instruction), the program stops.
    - The function returns the value in R_0.
    - So if R_0 contains b then

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

  - If the program never stops,

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

URM-Computable Functions

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

Example

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

$$\begin{array}{ll} 0: & \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \\ 1: & \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ 2: & \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0 \end{array}$$

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

Example

$$\begin{array}{ll} 0: & \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \\ 1: & \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ 2: & \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0 \end{array}$$

- 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

$$\begin{array}{ll} 0: & \texttt{if } \mathrm{R}_0 = 0 \texttt{ then goto } 3 \\ 1: & \mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1 \\ 2: & \texttt{if } \mathrm{R}_1 = 0 \texttt{ then goto } 0 \end{array}$$

- 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 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**.
- Total functions are functions which are defined for all elements of their domain.

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:

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

We derive a URM-program for it in several steps.
We follow the principle of successive refinement:

- We start with a high level program, which uses instructions which are not URM instructions.
  - High level is here only relative to URMs. The instructions we use are relative to normal programming languages low level.
- Then we replace this program in several steps by programs which are closer and closer to URM. Called **refinement**.
- Finally we arrive at a URM program.
- Correctness follows because first step was correct and each refinement step produced a program which was correct provided the previous program was correct.

Successive Refinement

Problem
↓

High Level Program
↓ (Refinement)

More concrete Program
↓ (Refinement)

...
↓ (Refinement)

Concrete URM Program

Example of URM-Comp. Function

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:

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

- 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 R1 = 0 then goto LabelEnd;
             R0 := R0 + 1;
             R1 := R1 ∸ 1;
             goto LabelBegin;
LabelEnd :
```


Example of URM-Comp. Function

```
LabelBegin : if R1 = 0 then goto LabelEnd;
             R0 := R0 + 1; R1 := R1 ∸ 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 R1 = 0 then goto LabelEnd;
             R0 := R0 + 1;
             R1 := R1 ∸ 1;
             if R2 = 0 then goto LabelBegin;
LabelEnd :
```


Example of URM-Comp. Function

```
LabelBegin : if R1 = 0 then goto LabelEnd;
             R0 := R0 + 1;
             R1 := R1 ∸ 1;
             if R2 = 0 then goto LabelBegin;
LabelEnd :
```

Step 5:

Resolve labels and obtain final program:

```
0 : if R1 = 0 then goto 4
1 : R0 := R0 + 1
2 : R1 := R1 ∸ 1
3 : if R2 = 0 then goto 0
```

III.2 (b) High Level Programming Constructs

- In this Subsection we will introduce some higher level program constructs for URMs, and how to translate them back into the original URM language.
- These constructs will be still be rather low level in terms of the theory of programming languages, but high enough in order to allow easily to introduce the programs needed in this module.

Convention Concerning Jump Addresses

- When inserting URM programs U as part of new URM programs, jump addresses will be adapted accordingly.
- E.g.in $\mathrm{R}_0 := \mathrm{R}_0 + 1$
  U
  $\mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1$
  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.

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
  $0:$ `if` $\mathrm{R}_0 = 0$ `then goto` 3
  $1:$ $\mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1$
  $2:$ `if` $\mathrm{R}_1 = 0$ `then goto` 0
- we write
  `LabelBegin` : $0:$ `if` $\mathrm{R}_0 = 0$ `then goto End`
  $1:$ $\mathrm{R}_0 := \mathrm{R}_0 \dot{-} 1$
  $2:$ `if` $\mathrm{R}_1 = 0$ `then goto LabelBegin`
  `End` :

Omitting Line Numbers

- We omit now the line numbers "k :" (referring to instructions $\mathrm{I}_k =$).
- Furthermore, labels don't have to start with `Label`, so we can write `Begin` instead of `LabelBegin`.
- We obtain the following program:

```
Begin : if R0 = 0 then goto End
        R0 := R0 ∸ 1
        if R1 = 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 R0 = 0 then goto End
        R0 := R0 ∸ 1
        if R1 = 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`.

while (x ≠ 0) do {···}

while (x ≠ 0) do {
  ⟨*Instructions*⟩};
stands for the following URM program:

```
LabelLoop :  if x = 0 then goto End;
             ⟨Instructions⟩
             goto LabelLoop;
```


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*⟩}

Repeat Loop

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.

Repeat Loop (More Direct Solution)

- We can translate as well

  repeat{
    ⟨*Instructions*⟩}
  until (x = 0);

  more directly as

```
LabelLoop :  ⟨Instructions⟩
             if x ≠ 0 then goto LabelLoop;
```

  which can be replaced by

```
LabelLoop :  ⟨Instructions⟩
             if x = 0 then goto End;
             goto LabelLoop;
```

x := 0

x := 0
stands for the following program:
while (x ≠ 0) do {x := x ∸ 1;};

y := x;

- 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.

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


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;                   −−x = 0; aux = x ∼
while (aux ≠ 0) do {
  aux := aux ∸ 1;
  x := x + 1;
  x := x + 1; };          −−x = x ∼ · 2; 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.

$y := x;$

- 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

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

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 ~; y = y ~
aux := z;
while (aux ≠ 0) do {
  aux := aux ∸ 1;
  x := x + 1; };           -- x = y ~ +z ~;
                           -- y = y ~; z = z ~; aux = 0;
```


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

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

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

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

```
x := y;
aux := z;
while (aux ≠ 0) do {
  aux := aux ∸ 1;
  x := x ∸ 1; };
```

Checking for Inequality

- We have
$$(x \dot{-} y) + (y \dot{-} x) \neq 0 \Leftrightarrow x \neq y$$
- Proof:
  - If $x > y$, then
$$\begin{array}{lcl} x \dot{-} y & > & 0\ , \\ y \dot{-} x & = & 0\ , \\ (x \dot{-} y) + (y \dot{-} x) & > & 0 \end{array}$$
  - If $y > x$, then
$$\begin{array}{lcl} y \dot{-} x & > & 0\ , \\ x \dot{-} y & = & 0\ , \\ (x \dot{-} y) + (y \dot{-} x) & > & 0 \end{array}$$

Checking for Inequality

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

- If $x = y$, then
$$\begin{array}{lcl} y \dot{-} x & = & 0\ , \\ x \dot{-} y & = & 0\ , \\ (x \dot{-} y) + (y \dot{-} x) & = & 0 \end{array}$$

Checking for Inequality

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

- So a while loop
$$\texttt{while}\ (x \neq y)\ \texttt{do}\ \{\cdots\}$$
can be replaced by
$$\texttt{while}\ ((x \dot{-} y) + (y \dot{-} x) \neq 0)\ \texttt{do}\ \{\cdots\}$$

Checking for Inequality

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

which can be replaced by

$$\begin{array}{l} \texttt{aux} := (x \dot{-} y) + (y \dot{-} x) \\ \texttt{while aux} \neq 0)\ \texttt{do} \\ \{\cdots \\ \ \texttt{aux} := (x \dot{-} y) + (y \dot{-} x) \\ \ \} \end{array}$$

If we unfold this further, we obtain the following:

while (x ≠ y) do {···}

Assume x, y denote different registers.

while (x ≠ y) do {
  ⟨*Statements*⟩};

stands for (aux, aux_i denote new registers):

$\text{aux}_0 := \text{x} \dot{-} \text{y}$;
$\text{aux}_1 := \text{y} \dot{-} \text{x}$;
$\text{aux} := \text{aux}_0 + \text{aux}_1$;
while (aux ≠ 0) do {
  ⟨*Statements*⟩
  $\text{aux}_0 := \text{x} \dot{-} \text{y}$;
  $\text{aux}_1 := \text{y} \dot{-} \text{x}$;
  $\text{aux} := \text{aux}_0 + \text{aux}_1$; };

Conclusion

- We have seen how to translate some higher level constructs into URMs.
- In terms of "real" programming languages we are still rather low level.
- We could however continue and in principle translate any standard high level language such as Java or Haskell into URMs.
  - It would in fact suffice to interpret a suitable machine language into URMs.
  - Since compilers translate high level languages into machine language this would show that high level languages can be translated into URMs.
- The goal of this chapter is to demonstrate that in principal we could translate any high level language into URMs, that URMs are really complete.

III.2 (c) URM-Computable Functions

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