CS_275 Automata and Formal Language Theory

Course Notes
Additional Material
(This material is no longer taught and not exam relevant)
Part III: Limits of Computation
Chapt. III.2: The URM

Anton Setzer

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

May 8, 2018

III.2 (a) Definition of the URM

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

III.2 (c) URM computable functions

III.2 (a) Definition of the URM

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

III.2 (c) URM computable functions

No Additional Material

For this subsection no additional material has been added yet.

III.2 (a) Definition of the URM

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

III.2 (c) URM computable functions

No Additional Material

For this subsection no additional material has been added yet.

III.2 (a) Definition of the URM

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

III.2 (c) URM computable functions

III.2 (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.

Definition 2.1

Definition

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

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

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

Remark

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

Notations for Partial Functions

Definition (Cont)

(d) *Assume*

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

Define

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

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

Notations for Partial Functions

Definition (Cont)

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

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

Notations for Partial Functions

Definition (Cont)

(e) *Assume*

$$\begin{array}{rcl} g & : & \mathbb{N}^k \xrightarrow{\sim} \mathbb{N} \ , \\ h & : & \mathbb{N}^{k+2} \xrightarrow{\sim} \mathbb{N} \ . \end{array}$$

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

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

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

Notations for Partial Functions

Definition (Cont)

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

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

by primitive recursion from a and h as follows:

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

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

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 : $\mathbb{N}^2 \to \mathbb{N}$, add$(x, y) := x + y$. We have

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

  Therefore

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

  where

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

  So add = primrec(g, h).

Addition (add)

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

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

Defining + from primrec in Haskell

In Haskell we can define add from primrec as follows

```
add :: Nat → Nat → Nat
add = primrec1 (λn → n) (λn m k → S k)
```


Examples for Primitive Recursion

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

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

  Therefore

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

  where

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

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

Multiplication (mult)

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

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

Examples for Primitive Recursion

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

  pred can be defined using primitive recursion:

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

  Therefore

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

  where

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

  So pred = 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

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

  So f = primrec(g, h).

Remark

- If f = primrec(g, h), then

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

- **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
$$f(\vec{n}, m)\!\uparrow \to f(\vec{n}, m+1)\!\uparrow \ .$$
- By induction it follows that $f(\vec{n}, m)\!\uparrow$ implies
$$\forall k \geq m. f(\vec{n}, k)\!\uparrow \ .$$

Example

- Let

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

- Let

$$f : \mathbb{N} \overset{\sim}{\to} \mathbb{N}\ , \quad f := \text{primrec}(1, h)\ ,$$
$$\text{i.e.} \quad f(0) \simeq 1\ , \quad f(n+1) \simeq h(n, f(n))\ .$$

- 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**.

$\vec{x}, \vec{y}$ etc.

- In many expressions we will have arguments, to which we don't refer explicitly.
  Example: Variables $x_0, \ldots, x_{n-1}$ in

$$f(x_0, \ldots, x_{n-1}, y) = \begin{cases} g(x_0, \ldots, x_{n-1}), & \text{if } y = 0, \\ h(x_0, \ldots, x_{n-1}), & \text{if } y > 0. \end{cases}$$

- We abbreviate $x_0, \ldots, x_{n-1}$, by $\vec{x}$.
- Then the above can be written shorter as

$$f(\vec{x}, y) = \begin{cases} g(\vec{x}), & \text{if } y = 0, \\ h(\vec{x}), & \text{if } y > 0. \end{cases}$$

- In general, $\vec{x}$ stands for $x_0, \ldots, x_{n-1}$, where the number of arguments n is clear from the context.

Examples

- If

$$f : \mathbb{N}^{n+1} \to \mathbb{N}$$

then in $f(\vec{x}, y)$,
$\vec{x}$ needs to stand for n arguments.
Therefore

$$\vec{x} = x_0, \ldots, x_{n-1}$$

- If

$$f : \mathbb{N}^{n+2} \to \mathbb{N}$$

then in $f(\vec{x}, y)$,
$\vec{x}$ needs to stand for $n+1$ arguments,
so

$$\vec{x} = x_0, \ldots, x_n$$

Examples

- If P is an $n+4$-ary relation, then in $P(\vec{x}, y, z)$, $\vec{x}$ stands for
$$x_0, \ldots, x_{n+1}$$
- Similarly, we write $\vec{y}$ for
$$y_0, \ldots, y_{n-1}$$
where n is clear from the context.
- Similarly for
$$\vec{z}, \vec{n}, \vec{m}, \ldots$$

Notation

- $$\forall \vec{x} \in \mathbb{N}.\varphi(\vec{x})$$
stands for
$$\forall x_0, \ldots, x_{n-1} \in \mathbb{N}.\varphi(x_0, \ldots, x_{n-1})$$
where the number of variables n is implicit (and usually unimportant).
- $$\exists \vec{x} \in \mathbb{N}.\varphi(\vec{x})$$
is to be understood similarly.

Notation

- $$\{\vec{x} \in \mathbb{N}^n \mid \varphi(\vec{x})\}$$
is to be understood as
$$\{(x_0, \ldots, x_{n-1}) \in \mathbb{N}^n \mid \varphi(x_0, \ldots, x_{n-1})\}$$
- $$\{(\vec{x}, y, z) \in \mathbb{N}^{n+2} \mid \varphi(\vec{x}, y, z)\}$$
is to be understood as
$$\{(x_0, \ldots, x_{n-1}, y, z) \in \mathbb{N}^{n+2} \mid \varphi(x_0, \ldots, x_{n-1}, y, z)\}$$
- Similar notations are to be understood analogously.

Notations for Partial Functions

Definition (Cont)

- *Let* $g : \mathbb{N}^{n+1} \stackrel{\sim}{\to} \mathbb{N}$.
We define $\mu y.(g(\vec{x}, y) \simeq 0)$
(Here $\vec{x}$ *stands for arguments* $x_1, \ldots, x_n$*).*

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

$\mu(g)$

Definition (Cont)

- *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

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

  Then

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

- Assume instead

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

  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 $f : \mathbb{N} \xrightarrow{\sim} \mathbb{N}$,
$f(0)\uparrow$,
$f(n) := 0$ 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 2.2

Lemma and Definition

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

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

The program can be defined so that it doesn't change $\mathtt{x}_0, \ldots, \mathtt{x}_{k-1}, \mathtt{z}_0, \ldots, \mathtt{z}_l$.
For U *we say it is* **a URM program which computes** $\mathtt{y} \simeq f(\mathtt{x}_0, \ldots, \mathtt{x}_{k-1})$ **and preserves** $\mathtt{z}_0, \ldots, \mathtt{z}_l$.

Intuition behind Lem. 2.2

- Lemma 2.2 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 $\mathtt{x}_0, \ldots, \mathtt{x}_{k-1}$, $\mathtt{z}_0, \ldots, \mathtt{z}_l$ 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:

$\mathrm{u}_0 := \mathrm{x}_0;$
$\ldots$
$\mathrm{u}_{k-1} := \mathrm{x}_{k-1};$
U'
$\mathrm{y} := \mathrm{u}_0;$

Lemma 2.3

Lemma

1. zero, succ *and* proj_i^n *are URM-computable.*
2. *If* $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$, $g_i : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ *are URM-computable, so is* $f \circ (g_0, \ldots, g_{n-1})$.
3. *If* $g : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$, *and* $h : \mathbb{N}^{n+2} \xrightarrow{\sim} \mathbb{N}$ *are URM-computable, so is the function* $f := \mathrm{primrec}(g, h)$ *defined by primitive recursion from* g *and* h.
4. *If* $g : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$ *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 2.3 (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_k^n is computed by the following program:
  $\mathrm{x}_0 := \mathrm{x}_k$.
  - Especially, if $k = 0$ then proj_k^n 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 2.3 (b)

Assume $f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N}$, $g_i : \mathbb{N}^k \xrightarrow{\sim} \mathbb{N}$ 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 2.2 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 2.3 (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 2.3 (b)

Let U' be defined as follows:

U_0
$\ldots$
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 2.3 (b)

U' is the program

U_0
$\ldots$
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 2.3 (b)

U' is the program

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

- **Case 2.1:** $f(g_0(\vec{x}), \ldots, g_{n-1}(\vec{x}))\uparrow$.
  V will loop, $\mathrm{U}'^{(k)}(\vec{x})\uparrow$, $f \circ (g_0, \ldots, g_{n-1})(\vec{x})\uparrow$.
- **Case 2.2:** Otherwise.
  The program reaches the end of program V and result in $x_0 \simeq f(g_0(\vec{x}), \ldots, g_{n-1}(\vec{x}))$.
  So $\mathrm{U}'^{(k)}(\vec{x}) \simeq (f \circ (g_0, \ldots, g_{n-1}))(\vec{x})$.

Proof of Lemma 2.3 (b)

In all cases

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

Proof of Lemma 2.3 (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

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

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 2.3 (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.
- $\ldots$
- Compute $f(\vec{n}, l)$ as $h(\vec{n}, l-1, f(\vec{n}, l-1))$, using the previous result.

Proof of Lemma 2.3 (c)

Plan for the program:

- Let $\vec{x} := x_0, \ldots, x_{n-1}$.
  Let y, z, u be new registers.
- Compute $f(\vec{x}, y)$ for $y = 0, 1, 2, \ldots, x_n$, and store result in z.
  - Initially we have $y = 0$ (holds for all registers except of $x_0, \ldots, x_n$ initially).
    We compute $z \simeq g(\vec{x})$ $(\simeq f(\vec{x}, 0))$.
    Then $y = 0$, $z \simeq f(\vec{x}, 0)$.

Proof of Lemma 2.3 (c)

- In step from y to $y + 1$:
  - Assume that we have $z \simeq f(\vec{x}, y)$.
  - We want that after increasing y by 1 the **loop invariant** $z \simeq f(\vec{x}, y)$ still holds.
    Obtained as follows
    - Compute $u \simeq h(\vec{x}, y, z)$ $(\simeq h(\vec{x}, y, f(\vec{x}, y)) \simeq f(\vec{x}, y + 1))$.
    - Execute $z := u$ $(\simeq f(\vec{x}, y + 1))$.
    - Execute $y := y + 1$.
    - At the end , $z \simeq f(\vec{x}, y)$ for the new value of y.
- Repeat this until $y = x_n$.
- Once y has reached x_n, z contains $f(\vec{x}, y) \simeq f(\vec{x}, x_n)$.
- Execute $x_0 := z$.

Proof of Lemma 2.3 (c)

Let

- U be a URM program, which computes $z \simeq g(\vec{x})$ and preserves y (by definition 2.2, it doesn't modify the arguments $\vec{x}$ of g);
- V be a program, which computes $u \simeq h(\vec{x}, y, z)$. (by definition 2.2, it doesn't change $\vec{x}, y, z$.)

Proof of Lemma 2.3 (c)

Let U' be as follows:

```
U                          -- Compute z ≃ g(x⃗)(≃ f(x⃗, 0))
while (x_n ≠ y) do {
  V                        -- Compute u ≃ h(x⃗, y, z)
                           -- will be  ≃ h(x⃗, y, f(x⃗, y)) ≃ f(x⃗, y + 1)
  z := u;
  y := y + 1; };
x_0 := z;
```

Proof of Lemma 2.3 (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 2.3 (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 2.3 (d)

Assume

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

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.

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

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 2.3 (d)

Plan for the program:

- Compute $g(\vec{\mathrm{x}}, 0)$, $g(\vec{\mathrm{x}}, 1)$, ... 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 2.3 (d)

Let U compute

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

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

$$\begin{array}{l}
\mathtt{repeat}\{ \\
\quad \mathrm{U} \\
\quad \mathtt{y} := \mathtt{y} + 1; \} \\
\mathtt{until}\ (\mathtt{z} = 0); \\
\mathtt{y} := \mathtt{y} \dot{-} 1; \\
\mathtt{x}_0 := \mathtt{y};
\end{array}$$

Omit rest of proof.

Proof of Lemma 2.3 (d)

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

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

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

($\mathtt{y}'$ is the value of $\mathtt{y}$ before this iteration).
If the loop terminates, we have

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

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

Proof of Lemma 2.3 (d)

- Finally $\mathtt{y}$ is decreased by one.
- Then $\mathtt{y}$ is the least $\mathtt{y}$ s.t.

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

- $\mathtt{x}_0$ is then set to that value.