Models of Computation

Last Updated September 26th, 2023

1 Introduction

Chances are you've already encountered several models of computation. Indeed, programming languages (such as Python, C, Java, Haskell, and PROLOG) are models of computation, as are digital circuits and CPU's in particular. However, the study of theoretical computer science tends to benefit from models of computation that are as simple as possible. This is because theoretical computing problems tend to already offer a complex intellectual challenge without additionally bogging down the mind with a model of computation that yields programs that require a myriad of cases to analyze. Thus, we desire a model of computation to be as simple as possoble, yet still meet the computing requirements that are assumed by the problem under investigation. The following are some approaches to achieving a minimalist-style computing model.

Automata Here, a computation is viewed as a sequence of state changes. A computation begins with an initial state and a machine has a finite-state controller that determines the next based on the current state and the current data that is being read. Examples: Finite Automata, Pushdown Automata, Turing Machines.

Register Machines These models are inspired by the architecture of a CPU where the machine consists of a finite number of registers along with the ability to perform basic logical and arithmetic operations on the words stored in each register. Examples: Random-Access Machines (RAM's), Unlimited Register Machines (URM's).

Function Families This approach views the function as the basis for computation and defines rules for constructing functions that are deemed "computable". To be in the function family means to be definable based on the provided rules of construction. Examples: Primitive and General Recursice Functions, Church's Lambda Calculus.

Rewriting Systems These models are similar to automata but with both data and state being combined into a single string of symbols. Examples: Markov Normal Algorithms, Post Production Systems.

Concurrency These models allow for multiple computation threads to simultaneously occur. Examples: Boolean and quantum Circuits, Petri Nets, Cellular Automata.

1.1 Uniform versus Non-Uniform Models of Computation

Definition 1.1. A problem instance is said to be **effectively solvable** iff there is some deterministic step-by-step procedure for solving the problem. In modern terms, we say that one can write a computer program for solving the problem.

Definition 1.2. A model of computation is said to be **uniform** iff every instance of the model is designed to solve all instances of a particular effectively solvable problem. A model of computation is **non-uniform** iff solving all instances of an effectively solvable problem requires the use of two or more instances of the model.

Examples of uniform models include general-purpose programming languages, such as C and Python. An example of a non-uniform model is a Boolean circuit, since it can handle only a finite number of different inputs. But computing problems usually have an infinite number of instances, and so an infinite number of circuits are required to solve all problem instances.

This lecture introduces the URM register-machine model along with the primitive recursive and general recursive function families. URM's find use in computability theory because their programs are readily encodable as a single integer. Such an encoding is called a **Gödel number** and is fundamental to both the study of computability and complexity theory. Primitive and general recursive function find use because programs are often easily and succinctly expressed using functions. Both the URM and general recursive function models are examples of what is called a **general model of computation**, meaning that it is a model that is capable of computing any process whose output is obtained in a deterministic step-by-step fashion with respect to one or more inputs being fed into the process. Programming languages, such as C, Python, and Java, are also considered general models of computation.

Regardless of what computing model is being considered, in this and subsequent computability lectures we make the assumption that the purpose of an instance of the model is to compute a function that maps one or more nonnegative integers to a nonnegative integer. By making this assumption, we do not lose any generality since any instance of any problem, including the solution to that instance, can be encoded with one or more nonnegative integers.

Definition 1.3. $\mathcal{N} = \{0, 1, 2, \ldots\}$ denote the set of nonnegative integers.

Unary Function $f : \mathcal{N} \to \mathcal{N}$ means that, for any input $x \in \mathcal{N}$, f assigns x to some value $f(x) \in \mathcal{N}$.

Multivariate Function For $m \geq 1$, $f : \mathcal{N}^m \to \mathcal{N}$ means that for any input vector $(x_1, \ldots, x_m) \in \mathcal{N}^m$, f assigns it to some value $f(x_1, \ldots, x_m) \in \mathcal{N}$.

In computability theory it's important to allow for functions that may not be defined on all inputs.

Definition 1.4. A **partial function** is one that is undefined on zero or more of its inputs. A function that is defined on all of its inputs is said to be a **total** function. Note: all total functions are (technically speaking) partial since they are undefined on zero of their inputs.

Example 1.5. The function $f : \mathcal{N} \to \mathcal{N}$ defined by $f(n)$ equals the value m for which $m^2 = n$ is only defined for $n = 1, 4, 9, 16, 25, \ldots$ and is undefined for all other values of n that are not perfect squares.

f is a partial function since it is only defined on 1, 4, 9, ...

2 The Unlimited Register Machine

The **Unlimited Register Machine (URM)** first introduced by Shepherdson and Sturgis (See Chapter 2 of Nigel Cutland's "Computability"). The purpose of a URM is to compute an m-ary function $f : \mathcal{N}^m \to \mathcal{N}$, from the set of m-tuples of nonnegative integers to nonnegative integers.

To begin, a **register** is a memory component that is capable of storing a nonnegative integer of arbitrary size. Registers form the basis of URM's. Indeed, a URM M consists of

1. r registers $R_1, \ldots, R_r$,
2. a finite program $P = I_1, \ldots, I_s$ consisting of s instructions that are used for step-by-step manipulation of the registers, and
3. a **program counter**, denoted `pc`, that stores the index of the next program instruction to be executed.

A URM M takes as input m nonnegative integers $\vec{x} = x_1, \ldots, x_m$, performs a computation on this input, and outputs a nonnegative integer, denoted $M(\vec{x})$, that is ultimately stored in register 1.

Definition 2.1. A **machine configuration** for an r-register URM is an $(r + 1)$-dimensional tuple whose first r components equal the integers currently stored in registers $R_1, \ldots, R_r$, and whose final component, called the **program counter (`pc`)**, is the index of the next instruction.

Initial Configuration The **initial configuration** is

$$\sigma_0 = (x_1, \ldots, x_m, \underbrace{0, \ldots, 0}_{r-m}, 1),$$

where $(x_1, \ldots, x_m)$ is the URM input vector.

Final Configuration A **final configuration** is any configuration whose program counter exceeds $s = |P|$.

Computation A **computation** of M on input $\vec{x}$ is a (possibly infinte) sequence of machine configurations $\sigma_0, \sigma_1, \ldots$ for which

1. σ_0 is the initial configuration
2. σ_{k+1} is obtained from σ_k by executing instruction I_i, where i is the value of σ_k's program counter `pc`, and updating the value of M's registers accordingly.

We write $M(\vec{x}) \downarrow$ (respectively, $M(\vec{x}) \uparrow$) in case the computation of M on input $\vec{x}$ is finite (respectively infinite).

2.1 URM Instruction Set

The following is a description of the different types of URM instructions, and how each affects the current machine configuration.

Zero Z(i), $1 \le i \le r$, assigns 0 to register R_i: $R_i \leftarrow 0$.

Sum S(i), $1 \le i \le r$, increments by 1 the value stored in R_i: $R_i \leftarrow R_i + 1$.

Transfer T(i, j), $1 \le i, j \le r$, assigns to R_j the value stored in R_i: $R_j \leftarrow R_i$.

Jump J(i, j, k), $1 \le i, j \le r$, $1 \le k \le s$, has the effect of setting `pc` to k in case R_i and R_j store the same integer. Otherwise `pc` is incremented by one.

Example 2.2. Consider a URM M with $r = 3$ registers and the following program.

I1. $J(1, 2, 6)$

I2. $S(2)$

I3. $S(3)$

I4. $J(1, 2, 6)$

I5. $J(1, 1, 2)$

I6. $T(3, 1)$

The following is the sequence of configurations produced by the computation $M(9, 7)$.

σ_i	R_1	R_2	R_3	**pc**	**Instruction**
0	9	7	0	1	J(1,2,6)
1	9	7	0	2	S(2)
2	9	8	0	3	S(3)
3	9	8	1	4	J(1,2,6)
4	9	8	1	5	J(1,1,2)
5	9	8	1	2	S(2)
6	9	9	1	3	S(3)
7	9	9	2	4	J(1,2,6)
8	9	9	2	6	T(3,1)
9	2	9	2	7	n/a

What function is being computed? It is worth noting that the above program is said to be **standard form** since since the computation will always terminate with the program counter at $s + 1$, where s is the number of instructions. A program is not in standard form in case the program counter can ever be assigned a value that exceeds $s + 1$.

Definition 2.3. An m-ary function $f : \mathcal{N}^m \to \mathcal{N}$ is **URM-computable** iff there exists a URM M for which, for all $\vec{x} \in \mathcal{N}^m$,

1. if $f(\vec{x})$ is defined, then $M(\vec{x}) = f(\vec{x})$, and
2. if $f(\vec{x})$ is undefined, then $M(\vec{x}) \uparrow$.

If f is defined on all inputs, then it is called **total URM-computable**. Otherwise, it is called **partially URM-computable**. Note: when we say a function is partially computable, it still may be possible that it is total computable. In other words, totally computable implies partially computable, but the converse is not necessarily true.

Example 2.4. Show that the function $f(x, y) = x + y$ is URM-computable.

x	y	0
R_1	R_2	R_3

Solution.

1. $J(2,3,5)$
2. $S(3)$
3. $S(1)$
4. $J(1,1,1)$

R_3 counts up to y and each 1 added to R_3 corresponds with adding 1 to R_1.

Example 2.5. By designing an appropriate URM M, show that the function

$$f(x) = \begin{cases} \lfloor x/2 \rfloor & \text{if } x \text{ is even} \\ \uparrow & \text{otherwise} \end{cases}$$

is URM-computable. Show the computations $M(2)$ and $M(3)$.

Solution.

x	0	0
R_1	R_2	R_3

1. $J(1, 2, 6)$
2. $S(2)$
3. $S(2)$
4. $S(3)$
5. $J(1, 1, 1)$
6. $T(3, 1)$

each iteration add 2 to R_2 and compare R_2 with R_1. Also, add 1 to R_3

Definition 2.6. We have the following definitions.

1. A **predicate** function is any function $f : \mathcal{N}^m \to \{0, 1\}$ whose output values are either 0 or 1.
2. A total predicate function is said to be **URM-decidable** iff there is a URM program that computes (i.e. **decides**) f.
3. A total unary predicate function is often referred to as a "property of the nonnegative integers".

Example 2.7. The property of being even can be represented by the function

$$\text{Even}(x) = \begin{cases} 1 & \text{if } x \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

Example 2.8. Provide a URM M that proves that the predicate function Even(x) from the previous example is URM-decidable.

Solution.

1. S(3)
2. J(1, 2, 9)
3. J(1, 3, 12)
4. S(2)
5. S(2)
6. S(3)
7. S(3)
8. J(1, 1, 2)
9. Z(1) //Even
10. S(1)
11. J(1, 1, 13)

x			
R_1	R_2	R_3	R_4

R_2: counts up evenly

R_3: count up oddly

12. Z(1) //odd

3 Primitive Recursive Functions

URM's have the following advantages when studying the theory of computation:

1. as we'll see in the next lecture, it seems relatively easy to encode a URM program as an integer,
2. the configuration of a URM can be simply described with an $r + 1$ tuple of integers, and
3. at times a theorem will require a proof that warrants writing a general program, which at times can seem relatively easy task with the URM model.

On the other hand, writing URM programs for specific computable functions can get complicated in a hurry. Recursion plays a fundamental role in computation. Moreover, recursion often provides very elegant solutions to problems. Thus it would seem desirable to study a model of computation that features the art and beauty of recursion. Indeed, in this section we examine the *primitive recursive functions*.

Definition 3.1. Rather than relying on a machine model, we provide a recursive definition for the set of **primitive recursive** functions.

For the base case, the following **basic functions** are primitive recursive.

1. The **zero function** 0
2. The **successor function** $x + 1$
3. The **Identity function** $f(x) = x$ for any variable x.
4. The **projection functions** $U_i^n(x_1, \ldots, x_n) = x_i$, where $n \geq 1$, and $1 \leq i \leq n$.

The first recursive case in the definition of primitive recursive functions makes use of **function composition**. Namely, suppose $g(y_1, \ldots, y_m)$, $f_1(\vec{x}), \ldots, f_m(\vec{x})$ are all primitive recursive, then so is $g(f_1(\vec{x}), \ldots, f_m(\vec{x}))$.

Finally, the second recursive case makes use of **recursion**. Namely, suppose $f(\vec{x})$ and $g(\vec{x}, y, z)$ are primitive recursive, then $h(\vec{x}, y)$ is primitive recursive, where

1. $h(\vec{x}, 0) = f(\vec{x})$, and
2. $h(\vec{x}, y + 1) = g(\vec{x}, y, h(\vec{x}, y))$

In the next several examples, we show that a given function is primitive recursive.

Example 3.2. $x + y$.

a. $x + 0 = x$.

b. $x + (y + 1) = (x + y) + 1$. (Apply the successor function to $x + y$)

Example 3.3. xy.

a. $x \cdot 0 = 0$.

b. $x(y + 1) = xy + x$.

→ basic function

Base case $y=0$

Add (xy, x)

```
// output x·y
mult(int: x, int y)
{
    if (y == 0)
        return 0;
    return (mult(x, y-1) + x);
}
```

$5 \cdot 2 = 10$

$5 \cdot 1 + 5 = 10$

$5 \cdot 0 + 5 = 5$

Example 3.4. x^y.

$\bar{1}(x) = 1 = 0 + 1$

$\bar{2}(x) = 2 = ((0 + 1) + 1)$

a. $x^0 = 0 + 1 = 1$. (1 is defined as the successor of 0)

b. $x^{y+1} = x^y \cdot x$. $= \mathrm{Mult}(x^y, x)$

Example 3.5. $x - 1$, where $0 - 1 =_{\text{def}} 0$.

a. $0 - 1 = 0$.

b. $(x + 1) - 1 = x$.

Sub1(0) = 0

Sub1(x+1) = x

Example 3.6. $x \dot{-} y$, where $x \dot{-} y =_{\text{def}} 0$ in case $y > x$.

$x \dot{-} y$

Solution.

$x \dot{-} 0 = x$

$x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 = \text{Subl}(x \dot{-} y)$

Subl function

Example 3.7. $\text{Sgn}(x) = 0$ if $x = 0$. Otherwise, $\text{Sgn}(x) = 1$.

Solution. Recursive definition

Base Case: $\text{Sgn}(0) = 0$

Recursive Case: $\text{Sgn}(x+1) = 1 + 0 \cdot \text{Sgn}(x)$

Example 3.8. $\overline{\text{Sgn}}(x) = 1$ if $x = 0$. Otherwise, $\overline{\text{Sgn}}(x) = 0$

Solution.

Non Rec. : $\overline{\text{Sgn}}(x) = 1 \dot{-} \text{Sgn}(x) = \text{Sub}(1, \text{Sgn}(x))$

Recursive

Base Case: $\overline{\text{Sgn}}(0) = 1$

Recursive Case: $\overline{\text{Sgn}}(x+1) = 0$

Example 3.9. $\text{Dist}(x, y) = |x - y|$.

$\text{Dist}(2,5) = |2-5| = 3$

Solution.

$\text{Dist}(x,y) = (x \dot{-} y) + (y \dot{-} x)$

$\text{Dist}(2,5) = (2 \dot{-} 5) + (5 \dot{-} 2) = 0 + 3 = 3$

Example 3.10. $x!$.

Solution.

$$0! = 1$$

$$(x+1)! = (x+1)x! = \text{Mult}(x+1, x!)$$

Example 3.11. Min(x, y).

Solution.

Non Recursive:

$$\text{Min}(x,y) = (x \dot- y)y + (y \dot- x)x$$

$$\text{Min}(x,y) = \overline{\overline{\text{Sgn}}(x \dot- y)}\, y + \overline{\text{Sgn}}(x \dot- y)\, x \qquad \underline{x \le y}$$

$$\text{Min}(0,2) = 0 \cdot 2 + 1 \cdot 0 = 0$$

$$\text{Min}(3,3) = 0 \cdot 3 + 1 \cdot 3 = 3$$

$$\text{Min}(6,3) = 1 \cdot 3 + 0 \cdot 6 = 3$$

$$\text{Min}(x,y) = \cancel{x \;(y \dot- x)} \quad x \dot- (x \dot- y) \checkmark$$

Example 3.12. Max(x, y). $\equiv x + (y \dot{-} x)$

Solution.

$$\text{Max}(7,3) = 7 + (3 \dot{-} 7) = 7 + 0 = 7$$

$$\text{Max}(2,5) = 2 + (5 \dot{-} 2) = 5$$

$$\text{Max}(x,y) = \text{sgn}(x \dot{-} y)x + \overline{\text{sgn}}(x \dot{-} y)y$$

$$\text{Max}(7,3) = \text{sgn}(7 \dot{-} 3)7 + \overline{\text{sgn}}(7 \dot{-} 3)3 = \boxed{7}$$

$$\text{Max}(2,5) = \text{sgn}(2 \dot{-} 5)2 + \overline{\text{sgn}}(2 \dot{-} 5)5 = 0 \cdot 2 + 1 \cdot 5 = 5$$

$$\text{Max}(6,6) = 0 \cdot 6 + \overline{\text{sgn}}(0) \cdot 6 = \boxed{6}$$

A **PR-decidable predicate** $M(\vec{x})$ is a primitive recursive function whose range is $\{0, 1\}$. Predicate functions are often used to indicate whether or not an integer has some given property. For example, the predicate

$$M(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

indicates whether or not an integer x is even.

Note that we may view predicates as Boolean functions if we equate 1 with `true`, and 0 with `false`.

Example 3.13. Show that the predicate function $x < y$ is PR-decidable.

Solution.

$$LT(x,y) = sgn(y \dot{-} x)$$

Theorem 3.14. If $M_1(\vec{x}), \ldots, M_k(\vec{x})$ are PR-decidable predicates, and $f_1(\vec{x}), \ldots, f_k(\vec{x})$ are primitive recursive functions, and, for every $\vec{x}$, exactly one of $M_1(\vec{x}), \ldots, M_k(\vec{x})$ is true, then

$$g(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } M_1(\vec{x}) \text{ is true} \\ \vdots & \vdots \\ f_k(\vec{x}) & \text{if } M_k(\vec{x}) \text{ is true} \end{cases}$$

is a primitive recursive function.

Proof. The theorem is true since $M_1(\vec{x})f_1(\vec{x}) + \cdots + M_k(\vec{x})f_k(\vec{x})$ is primitive recursive. □

Theorem 3.15. If $P(\vec{x})$ and $Q(\vec{x})$ are PR-decidable predicates, then so are $\overline{P}(\vec{x})$, $P(\vec{x}) \wedge Q(\vec{x})$, $P(\vec{x}) \vee Q(\vec{x})$.

Proof. We have

$$\overline{P}(\vec{x}) = \text{Sub}(1, P(\vec{x})),$$

$$P(\vec{x}) \wedge Q(\vec{x}) = \text{Mult}(P(\vec{x}), Q(\vec{x})),$$

and

$$P(\vec{x}) \vee Q(\vec{x}) = \text{Max}(P(\vec{x}), Q(\vec{x})).$$

□

Example 3.16. x mod y is the remainder of x divided by y, where 0 mod $y = 0$ and x mod $0 = 0$.

5 mod 5 = 0

Solution.

6 mod 5 = 1   7 mod 5 = 2

8 mod 5 = 3   9 mod 5 = 4

10 mod 5 = 0

Recursion on y.

$0 \bmod y = 0$ (Base Case)

$$(x+1) \bmod y = \begin{cases} 0 & \text{if } y = 0 \\ (x \bmod y)+1 & \text{if } (x \bmod y) \neq y \dot{-} 1 \wedge y \neq 0 \\ 0 & \text{if } (x \bmod y) = y \dot{-} 1 \wedge y \neq 0 \end{cases}$$

Example 3.17. x/y is the integer quotient of x divided by y, where $x/0 = 0$.

Solution.

$5/5 = 1 \quad 6/5 = 1 \quad 7/5 = 1 \quad 8/5 = 1$

$9/5 = 1 \quad 10/5 = 2$

Recursive Definition (on x)

Base Case: $0/y = 0$

Recursive Case

$$x+1/y = \begin{cases} 0 & \text{if } y = 0 \\ x/y & \text{if } x \bmod y \neq y \dot{-} 1 \wedge y \neq 0 \\ (x/y)+1 & \text{if } x \bmod y = y \dot{-} 1 \wedge y \neq 0 \end{cases}$$

Example 3.18. $\text{Div}(x, y) = 1$ if y divides evenly into x. Otherwise, $\text{Div}(x, y) = 0$. Note: assume 0 divides 0, but does not divide any positive integers.

Solution.

$$\text{Div}(x,y) = \overline{\text{sgn}}(x \bmod y) \wedge y \neq 0.$$

3.1 Bounded Primitive Recursive Iterators

An **iterator** is a function that makes use of at least one **index variable** an iterates over the domain of this variable in order to compute the function output. A **bounded iterator** iterates over the index-variable domain antil an upper bound is reached is reached. In this section we show that three iterators commonly used in practice (bounded sum, product, and least satisfying) are primitive recursive.

Theorem 3.19. If $f(\vec{x}, z)$ is primitive recursive, then so is the **bounded sum** function

$$\sum_{z=0}^{y} f(\vec{x}, z) = f(\vec{x}, 0) + \cdots + f(\vec{x}, y).$$

Proof. First notice that z represents an **index variable**, and that the bounded sum does not depend on z, but rather on $\vec{x}$ and y. Thus, let

$$h(\vec{x}, y) = \sum_{z=0}^{y} f(\vec{x}, z).$$

Then we have the following recursive definition for h.

Note: an alternative notation for bounded sum is

$$\sum_{z \leq y} f(\vec{x}, z).$$

a. $h(\vec{x}, 0) = f(\vec{x}, 0)$, which is primitive recursive, since $g(\vec{x}) = f(\vec{x}, 0)$ is PR by composition.

b. $h(\vec{x}, y + 1) = h(\vec{x}, y) + f(\vec{x}, y + 1)$.

Therefore, h is PR. □

Note: **bounded product** function

$$\prod_{z=0}^{y} f(\vec{x}, z) = f(\vec{x}, 0) \cdot \cdots \cdot f(\vec{x}, y)$$

may be similarly defined.

Theorem 3.20. If $f(\vec{x}, z)$ is primitive recursive predicate, then the **least satisfying function**

$$\underset{z \le y}{\lambda} f(\vec{x}, z) = \begin{cases} \text{least } z \text{ for which } f(\vec{x}, z) = 1 & \text{if such } z \text{ exists} \\ y + 1 & \text{otherwise} \end{cases}$$

is primitive recursive, where i.e the least $z < y$ for which the statement $f(\vec{x}, z) = 0$ evaluates to true, or y if no such z exists.

The least satisfying function acts like a `for` loop, where $f(\vec{x}, y)$ is a condition for breaking out of the loop before z is assigned $y + 1$.

For example, the following procedural code computes $\underset{z \le y}{\lambda} f(\vec{x}, z)$.

```
for(z=0; z <= y && !f(x,z); z++);

return z;
```

Proof. Again notice that z is an index variable, and the function only depends on $\vec{x}$ and y.

We claim that

$$\underset{z \le y}{\lambda} f(\vec{x}, z)$$

is equivalent to the $\mathcal{PR}$ function

$$\sum_{i=0}^{y} \prod_{j=0}^{i} \mathrm{Sgn}(1 - f(\vec{x}, j)).$$

zero once there is a j for which $f(\vec{x}, j) = 1$

Case 1. $f(\vec{x}, j) = 0$ for all $j \le y$. Then

$$\sum_{i=0}^{y} \prod_{j=0}^{i} \mathrm{Sgn}(1 - f(\vec{x}, j)) =$$

$$\sum_{i=0}^{y} \prod_{j=0}^{i} 1 = \sum_{i=0}^{y} 1 = y + 1 = \underset{z \le y}{\lambda} f(\vec{x}, z).$$

Case 2. There is a least $z \le y$, for which $f(\vec{x}, z) = 1$. Then

$$\sum_{i=0}^{y} \prod_{j=0}^{i} \mathrm{Sgn}(1 - f(\vec{x}, j)) =$$

$$\sum_{i=0}^{z-1} \prod_{j=0}^{i} \mathrm{Sgn}(1 - f(\vec{x}, j)) + \sum_{i=z}^{y} \prod_{j=0}^{i} \mathrm{Sgn}(1 - f(\vec{x}, j)) =$$

$$\sum_{i=0}^{z-1} 1 + \sum_{i=z}^{y-1} 0 = z = \underset{z \le y}{\lambda} f(\vec{x}, z).$$

□

Example 3.21. Use bounded least-satisfying to prove that $f(x) = \lfloor \sqrt[3]{x} \rfloor$ is a $\mathcal{PR}$ function.

Solution.

$$\lfloor \sqrt[3]{15} \rfloor =$$

$$2 < \sqrt[3]{15} < 3 \Rightarrow$$

$$\lfloor \sqrt[3]{15} \rfloor = 2$$

$$\lfloor \sqrt[3]{x} \rfloor = \left[\lambda_{z \le x} (z^3 > x) \right] \dot{-} 1$$

3.2 Unbounded Least Satisfying

Notice that every primitive recursive function is total computable. This can be proved by structural mathematical induction over the set of primitive-recursive functions. Thus, we can conclude that not all URM-computable functions are primitive recursive, since some URM-computable functions are not total. Therefore, apparently we need more techniques for defining at least all the URM-computable functions. It turns out that we need exactly one additional technique, called *unbounded least satifying*.

Definition 3.22. Let $f(\vec{x}, y)$ be a predicate function. Then the **unbounded least satisfying** function, denoted

$$\underset{y}{\lambda} f(\vec{x}, y),$$

evaluates to the least y for which the predicate function $f(\vec{x}, 0) = 0, f(\vec{x}, 1) = 0, \ldots, f(\vec{x}, y-1) = 0$ are all defined, and $f(\vec{x}, y) = 1$". If no such y exists, then

$$\underset{y}{\lambda} f(\vec{x}, y),$$

is undefined.

We see that unbounded minimalization has the effect of a `while` loop that may never terminate in case either i) the computation of $f(\vec{x}, y)$ does not terminate for some y, or ii) $f(\vec{x}, y)$ is always zero.

Definition 3.23. Recall Definition 3.1 that recursively defines the set $\mathcal{PR}$ of Primitive Recursive functions. If we use the same definition with the exception of i) replacing the family name with $\mathcal{GR}$, for the set of **General Recursive functions**, and ii) add the additional recursive case:

> "If $f(\vec{x}, y)$ is a general-recursive predicate function, then so is
>
> $$\underset{y}{\lambda} f(\vec{x}, y)\text{"},$$

Then we have the definition for the set $\mathcal{GR}$ of general recursive functions.

Example 3.24. Use unbounded least satisfying to prove that

$$f(x) = \begin{cases} \lfloor x/2 \rfloor & \text{if } x \text{ is even} \\ \uparrow & \text{otherwise} \end{cases}$$

is a general recursive (GR) function.

Solution.

$$\left[\lambda_y (2y = x) \right]$$

Theorem 3.25. An m-ary function $f : \mathcal{N}^m \to \mathcal{N}$ is URM-computable iff it is general recursive.

The proof of Theorem 3.25 requires two directions. First we must show that an arbitrary general recursive function is URM-computable. This amounts to showing the following.

$GR \subseteq URM$

1. Each $\mathcal{PR}$ basic function is URM computable.
2. The composition of two or more URM-computable functions is also URM computable.
3. If two URM-computable functions f and g are used to recursively define a computable function h, then h is also URM computable.
4. If predicate function $f(x, y)$ is URM computable, then so is

$$\underset{y}{\lambda} f(\vec{x}, y).$$

Since the use of basic functions, composition, recursion, and unbounded least satisfying are the only tools one can use to define a $\mathcal{GR}$ function, from the above four statements it follows that every definable GR function must also be URM computable. We provide informal programs (that can readily be converted to URM programs) on the next two pages that establish statements 2, and 3. For statement 1, and for each basic function, the reader is asked to provide a URM program that computes it. For statement 4, the reader is asked to provide an informal program (similar to the ones provided for statements 2 and 3) that establishes the URM computability of the unbounded least satisfying function.

The second part of the proof seems more challenging: showing that any URM-computable function is in fact a GR function. This will require the further development of $\mathcal{PR}$ functions in a later lecture.

In what follows, we say that a URM register is **safe** if it is not referenced by any URM instruction or has been (informally) designated for storing some other previously mentioned quantity.

An Informal URM Program that Supports Composition of Functions (Statement 2)

Let $P_1, \ldots, P_m$ be URM programs for computing $f_1(\vec{x}), \ldots, f_m(\vec{x})$, respectively, where we assume that $\vec{x} = (x_1, \ldots, x_n)$ is an n-ary vector of natural-number inputs. Let Q be a URM program that computes $g(y_1, \ldots, y_m)$. Then the following informal program can be used to define a URM program for computing $g(f_1(\vec{x}), \ldots, f_m(\vec{x}))$.

Input $x_1, \ldots, x_n$.

Copy $x_1, \ldots, x_n$ to safe registers.

For each $i = 1, \ldots, m$,

> Copy $x_1, \ldots, x_n$ to registers $R_1, \ldots, R_n$, respectively.
>
> Execute P_i's instructions.
>
> Copy the output y_i in R_1 to a safe register.
>
> Clear all registers used by P_i.

Copy $y_1, \ldots, y_m$ to registers $R_1, \ldots, R_m$, respectively.

Execute Q's instructions.

Return R_1.

An Informal URM Program that Supports Recursion (Statement 3)

Recall the definition of recursion.

1. $h(\vec{x}, 0) = f(\vec{x})$, and
2. $h(\vec{x}, y+1) = g(\vec{x}, y, h(\vec{x}, y))$

Assume that $\vec{x} = (x_1, \ldots, x_n)$ is an n-ary vector of natural-number inputs. Let URM program P compute $f(x_1, \ldots, x_n)$. Let URM program Q compute $g(x_1, \ldots, x_n, y, z)$.

Input $x_1, \ldots, x_n, y$.

Copy $x_1, \ldots, x_n$ and y to safe registers.

$R_{n+1} \leftarrow 0$. //Clear this register before executing P.

Execute P's instructions.

If $y = 0$, then Return R_1. //Base case

Copy $R_1 = f(x_1, \ldots, x_n)$ to a safe register $\hat{R}$.

Clear all registers used by P.

Designate a safe register that is to store the value of counter c initialized as 0.

For each $c = 0, \ldots, y - 1$,

> Copy $x_1, \ldots, x_n$ and y to registers $R_1, \ldots, R_n, R_{n+1}$, respectively.
> $R_{n+2} \leftarrow \hat{R}$. //$R_{n+2}$ now holds $h(\vec{x}, c)$
> Execute Q's instructions.
> $\hat{R} \leftarrow R_1$. //$\hat{R}$ now holds $h(\vec{x}, c+1)$
> Clear all registers used by Q.

Return $\hat{R}$. //$\hat{R}$ now holds $h(\vec{x}, y)$, the desired output.

Exercises

Note: for each exercise you may use all lecture examples, theorems, and previous exercises to establish that a function is primitive/general recursive.

For exercises 1-5 you may find it useful and fun to test your solutions with an online URM simulator:

`https://sites.oxy.edu/rnaimi/home/URMsim.htm`

1. Provide URM-programs that compute the following functions.

   a.
   $$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \end{cases}$$

   b. $f(x) = 4$

   c.
   $$f(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{if } x > y \end{cases}$$

2. Show that the function
   $$f(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$
   is URM-computable.

3. Show that the function $f(x, y) = \min(x, y)$ is URM-computable.

4. Suppose $f(x)$ and $g(x)$ are both URM-computable via programs P_1 and P_2 respectively. Provide an outline of a URM program that computes $f(g(x))$.

5. Suppose P_1 and P_2 are two programs, and we desire to make a third program P_3 whose behavior can be described as "Run P_1 until it halts. Then run P_2 on the final register configuration produced by P_1." Explain why P_1P_2 may not have the desired effect, where P_1P_2 means list the instructions of P_2 immediately after those of P_1. Explain the alterations that may need to be made in order for P_1P_2 to work as desired.

6. If $f(x)$ is URM-computable via a program that has no jump instructions, then prove that $f(x) = C$ of $f(x) = x + C$, for some constant $C \in \mathcal{N}$.

7. Using the definition of a PR function and Examples 3.2 to 3.11, show that the binary relations $= (x, y)$, $(\neq (x, y)$, $< (x, y)$, $\leq (x, y)$, $> (x, y)$, $\geq (x, y)$ are all PR-decidable predicates. For example, $= (x, y)$ returns 1 if $x = y$, and returns 0 otherwise.

8. Using the definition of a PR function and Examples 3.2 to 3.11, show that $\text{Even}(x)$ is PR, where $\text{Even}(x) = 1$ iff x is even. Do the same for $\text{Odd}(x)$.

9. Using the definition of a PR function and Examples 3.2 to 3.11, show that $\text{Min3}(x, y, z)$ is PR.

10. If $M(\vec{x}, z)$ is a PR-decidable predicate, then show that the **bounded universal quantifier** function

$$\forall_{z \le y} M(\vec{x}, z)$$

is also a PR-decidable predicate, where

$$\forall_{z \le y} M(\vec{x}, z)$$

evaluates to 1 iff $M(\vec{x}, 0) = M(\vec{x}, 1) = \cdots = M(\vec{x}, y) = 1$.

11. If $M(\vec{x}, z)$ is a PR-decidable predicate, then show that the **bounded existential quantifier** function

$$\exists_{z \le y} M(\vec{x}, z)$$

is also a PR-decidable predicate, where

$$\exists_{z \le y} M(\vec{x}, z)$$

evaluates to 1 iff $M(\vec{x}, z) = 1$ for some $z \le y$.

12. Prove that the following functions are primitive recursive.

    a. $D(x)$ equals the number of divisors of x. Hint: $D(0) = 1$.

    b. $\text{Prime}(x)$ is the predicate function that evaluates to 1 iff x is a prime number.

    c. p_x denotes the function that, on input x, returns the x th prime number. Here we assume $p_0 = 0$, $p_1 = 2$, $p_2 = 3$, etc..

    d. $(x)_y$ is a function of x and y and returns the exponent of p_y in the prime factorization of x. For example $(24)_1 = 3$ since the first prime number is 2, and 2^3 is in the prime factorization of 24. We assume $(x)_y = 0$ in case either $x = 0$ or $y = 0$.

    e. $\lfloor \sqrt{x} \rfloor$.

    f. $\text{LCM}(x, y)$ equals the least common multiple of x and y.

    g. $\text{GCD}(x, y)$ equals the greatest common divisor of x and y. Hint: $\text{GCD}(0, 0) = 0$.

    h. $\text{PD}(x)$ equals the number of prime divisors of x.

    i. $\phi(x)$ equals the number of positive integers less than x that are relatively prime to x.

13. Prove that any polynomial function $p(x) = a_n x^n + \cdots + a_1 x + a_0$ is primitive recursive.

14. Let $\pi(x, y) = 2^x(2y+1)-1$. Prove that π is total computable, and is a one-to-one correspondence between $\mathcal{N}^2$ and $\mathcal{N}$. Also, show that both π_1 and π_2 are primitive recursive, where $\pi(\pi_1(x), \pi_2(x)) = x$.

15. Show that the following problems are PR-decidable.

    a. $M(x) = 1$ iff x is odd.

    b. $M(x) = 1$ iff x is a power of a prime number.

    c. $M(x) = 1$ iff x is a perfect cube.

16. Show that $x/2^y$ is primitive recursive.

17. Define $\alpha(i, x)$ as the function that returns the i th bit in the binary representation of x. Prove that $\alpha(i, x)$ is primitive recursive. For example $\alpha(0, 2) = 0$, $\alpha(1, 2) = 1$, while $\alpha(i, 2) = 0$ for all $i \geq 2$.

18. Let $\text{Len}(x)$ be the function that returns the length of the binary representation of x. Prove that $\text{Len}(x)$ is primitive recursive. Note: $\text{Len}(0) = 1$.

19. Show that the following function is general recursive.

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ \uparrow & \text{otherwise} \end{cases}$$

20. Show that the following function is general recursive.

$$f(z) = \begin{cases} 1 & \text{if } \exists x \exists y (z = 17x^3 - 29x^2y^2 + 37x^2 - 41y^2 + 31x + 1331) \\ \uparrow & \text{otherwise} \end{cases}$$

where x and y are variables for which $\text{dom}(x) = \text{dom}(y) = \mathcal{N}$. Hint: use Exercise 14.

21. Let $f(\vec{x}, z)$ be a primitive recursive predicate. The **bounded parity** function is defined as

$$\bigoplus_{z \leq y} f(\vec{x}, z) = f(\vec{x}, 0) \oplus f(\vec{x}, 1) \oplus \cdots \oplus f(\vec{x}, y),$$

and equals the parity of the binary string

$$f(\vec{x}, 0) \cdot f(\vec{x}, 1) \cdot \cdots \cdot f(\vec{x}, y).$$

Use recursion (on variable y) and one or more PR functions from this chapter to show that bounded parity is primitive recursive.

22. Let $\text{Trunc}(x, i)$ denote the number x with its first i digits cut off. For example, $\text{Trunc}(958, 0) = 958$, $\text{Trunc}(958, 1) = 95$, $\text{Trunc}(958, 2) = 9$, and $\text{Trunc}(958, i) = 0$ for every $i \geq 4$. Use recursion (on variable i) and one or more PR functions from this chapter to show that Trunc is primitive recursive.

Exercise Solutions

1. Provide URM-programs that compute the following functions.

   a. $J(1,2,3), S(2), T(2,1)$

   b. $Z(1), S(1), S(1), S(1), S(1)$.

   c. $J(1,3,5), J(2,3,6), S(3), J(1,1,1), S(4), T(4,1)$

2. 1. $J(1,2,10)$, 2. $T(1,3)$, 3. $T(2,4)$, 4. $S(3)$, 5. $J(2,3,10)$, 6. $S(4)$, 7. $S(5)$, 8. $J(1,4,12)$, 9. $J(1,1,4)$, 10. $Z(1)$, 11. $J(1,1,15)$, 12. $T(5,1)$, 13. $J(1,1,14)$

3. 1. $J(1,2,10)$, 2. $T(1,3)$, 3. $T(2,4)$, 4. $S(3)$, 5. $J(2,3,10)$, 6. $S(4)$, 7. $J(1,4,9)$, 8. $J(1,1,4)$, 9. $T(2,1)$,

4. First execute the instructions of P_2. Let m be the index of the maximum register used by P_2. Next, perform the instructions $Z(2), \ldots, Z(m)$. Finally, execute the instructions of P_1.

5. Suppose P_1 has k instructions, then any jump instruction of P_1 that jumps to a value $v > k$, should now jump to $k+1$, so that the first instruction of P_2 executes next. Furthermore, each jump instruction of P_2 should have its jump address incremented by k so that jumps do not accidentally land back in P_1.

6. Case 1: register R_1 is written over via either a $Z(1)$ or $T(m,1)$ instruction, for some $m > 1$. In case of a $Z(1)$ instruction, R_1 can hold at most a constant C which equals the number of $S(1)$ instructions that follow the final $Z(1)$ instruction. In case R_1 was written over via a transfer from register R_m, R_1 equals C, where C is the number of $S(m)$ instructions that precede the final $T(m,1)$ instruction, plus the number of $S(1)$ instructions that follow the final $T(m,1)$ instruction.

   Case 2: register R_1 is never written over. Then R_1 will hold the value $x + C$, where C is the number of $S(1)$ program instructions.

   If $f(x)$ is URM-computable via a program that has no jump instructions, then prove that $f(x) = C$ of $f(x) = x + C$, for some constant $C \in \mathcal{N}$.

7. We have the following.

   a. $(x < y) = \text{sgn}(y - x)$.

   b. $(x > y) = \text{sgn}(x - y)$.

   c. $(x = y) = \overline{\text{sgn}}(x - y) \wedge \overline{\text{sgn}}(y - x)$.

   d. $(x \leq y) = 1 - (x > y) = \overline{\text{sgn}}(x - y)$.

   e. $(x \geq y) = 1 - (x < y) = \overline{\text{sgn}}(y - x)$.

8. **Base case.** $\text{Even}(0) = 1$. **Recursive case.** $\text{Even}(x+1) = 1 - \text{Even}(x)$. $\text{Odd}(x)$ is defined similarly, but with the base case $\text{Odd}(0) = 0$.

9. $\text{Min3}(x, y, z) = \text{Min}(\text{Min}(x, y), z)$.

10. We have
$$\underset{z\le y}{\forall} M(\vec{x}, z) = \prod_{z\le y} M(\vec{x}, z).$$
Therefore, the function is primitive recursive since M and bounded product are primitive recursive.

11. We have
$$\underset{z\le y}{\exists} M(\vec{x}, z) = \text{sgn}(\sum_{z\le y} M(\vec{x}, z)).$$
Therefore, the function is primitive recursive since M, bounded sum, and `sgn` are all primitive recursive.

12. The following functions are primitive recursive.

   a. We have
$$D(x) = \sum_{z\le x} \text{Div}(z, x),$$
   which is primitive recursive since both bounded sum and `Div` are primitive recursive.

   b. We have
$$\text{Prime}(x) = (x \ge 2) \wedge (D(x) = 2).$$

   c. Using recursion, we have $p_0 = 0$,
$$p_{x+1} = \underset{z\le p_x!+1}{\lambda} (z > p_x \wedge \text{Prime}(z)).$$

   d. We have $(0)_i = 0$ for all i. For $x \ge 1$ we have
$$(x)_i = \underset{z\le x}{\lambda} \neg\text{Div}(p_i^z, x) - 1.$$

   e. We have
$$\lfloor\sqrt{x}\rfloor = \underset{z\le x+1}{\lambda} (z^2 > x) - 1.$$

   f. We have
$$\text{LCM}(x, y) = \begin{cases} 0 & \text{if } x = 0 \vee y = 0 \\ \underset{z\le xy}{\lambda} (\text{Div}(x, z) \wedge \text{Div}(y, z) \wedge z > 0) & \text{if } x > 0 \wedge y > 0 \end{cases}$$

   g. We have
$$\text{GCD}(x, y) = \begin{cases} 0 & \text{if } x = 0 \wedge y = 0 \\ y & \text{if } x = 0 \wedge y > 0 \\ x & \text{if } x > 0 \wedge y = 0 \\ xy/\text{LCM}(x, y) & \text{if } x > 0 \wedge y > 0 \end{cases}$$
   The last case makes use of the identity $\text{LCM}(x, y)\text{GCD}(x, y) = xy$.

   h. We have
$$\text{PD}(x) = \begin{cases} 0 & \text{if } x \le 1 \\ \sum_{z\le x} \text{Div}(z, x) \wedge \text{Prime}(z) & \text{if } x \ge 2 \end{cases}$$

i. We have

$$\phi(x) = \begin{cases} 0 & \text{if } x = 0 \\ \sum_{z \le x}(\mathrm{GCD}(z, x) = 1) & \text{if } x \ge 1 \end{cases}$$

13. We use induction. For $n = 0$, $p(x) = a_0$ is a constant, and hence PR by composing the successor function with itself a_0 times. Now assume that any nth-degree polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$ is PR, for some $n \ge 0$. Consider the $(n+1)$-degree polynomial $q(x) = a_{n+1}x^{n+1} + a_n x^n + \cdots + a_1 x + a_0$. Then we can rewrite $q(x)$ as

$$q(x) = x(a_{n+1}x^n + a_n x^{n-1} + \cdots + a_1) + a_0 = xp(x) + a_0,$$

where $p(x) = a_{n+1}x^n + a_n x^{n-1} + \cdots + a_1$ is an n th-degree polynomial, and hence PR by the inductive assumption. Therefore, $q(x)$ is PR, since it is the sum of a_0 with the PR $x \cdot p(x)$.

14. π is primitive recursive since it only uses the power, addition, and multiplication functions, all of which have been shown to be primitive recursive. To see that π maps onto the nonnegative integers, consider natural number n, then we need an x and y for which $\pi(x, y) = n$, i.e.

$$2^x(2y + 1) = n + 1.$$

But $n + 1$ is a positive integer, and every positive integer has a unique prime factorization. Thus, there is a unique x for which i) 2^x divides $n+1$ and for which $(n+1)/2^x$ is odd, meaning that there is a unique y for which $2y + 1 = (n + 1)/2^x$. Hence, π maps onto $\mathcal{N}$ and, by the uniqueness of x and y, we see that the mapping is also one-to-one. Therefore, π is a primitive recursive bijection.

To see that π_1 and π_2 are primitive recursive, note that $\pi_1(n) = (n + 1)_1$, while

$$\pi_2(n) = (((n + 1)/2^{\pi_1(n)}) - 1)/2.$$

15. The following are all PR-decidable predicates.

    a. $\neg\mathrm{Div}(2, x)$.

    b. $x \ge 2 \wedge \underset{z \le x}{\exists}(x = p_z^{(x)_z})$.

    c. $\underset{z \le x}{\exists}(z^3 = x)$.

16. $x/2^i$ is primitive recursive since both division and power are primitive recursive.

17. To obtain $\alpha(i, x)$ we can divide x by 2^i which has the effect of shifting x to the right by i bits, and so the i th bit of x is now bit zero of $x/2^i$. Therefore,

$$\alpha(i, x) = x/2^i \bmod 2.$$

18. Define $\lfloor \log x \rfloor$ by

$$\lfloor \log x \rfloor = \underset{z \le x}{\lambda}(2^z > x) - 1.$$

Then $\mathrm{Len}(x) = \lfloor \log x \rfloor + 1$. You may want to try some different values of x to verify this.

19.

20.

21. We have the following. **Base case:**

$$\bigoplus_{z \leq 0} f(\vec{x}, z) = f(\vec{x}, 0)$$

is PR since f is assume PR.

Recursive case:

$$\bigoplus_{z \leq y+1} f(\vec{x}, z) = \left(\bigoplus_{z \leq y} f(\vec{x}, z) \right) \oplus f(\vec{x}, y+1).$$

22. We have the following. **Base case:** $\mathrm{Trunc}(x, 0) = x$. **Recursive case:** $\mathrm{Trunc}(x, i+1) = \mathrm{Trunc}(x, i)/10$.