Computation; Notes

July 29, 2020

Contents

1 Computable Functions

1.1 Basic Concepts

Partial Functions

A *partial function* generalizes the usual definition of function, the idea being that this kind of function is potentially not defined on the entire domain. Formally:

Definition 1.1. A *partial function* f from X to Y (written as $f : X \nrightarrow Y$) is a triple (g, X, Y) such that $X' \subseteq X$ and $g : X' \to Y$ is a function. Furthermore:

- The *domain* of f is denoted by $\mathsf{Dom}(f)$ and is equal to X';
- If $\mathsf{Dom}(f) = X$ then f is a *total function*[1];
- If $x \in (X \setminus \mathsf{Dom}\, f)$ then $f(x)$ is said to be *undefined*, denoted $f(x) = -$, on the other hand, if $x \in \mathsf{Dom}\, f$ then we write $f(x) = y$ with $y = g(x)$ and say that f is *defined* at x.

Henceforth the word "function" will always mean "partial function." As an example, consider the (partial) function:

$$\begin{aligned} f : \mathbb{N}_0 &\nrightarrow \mathbb{N}_0 \\ n &\mapsto \sqrt{n}. \end{aligned}$$

If $n \in \mathbb{N}_0$ is not a perfect square, then $f(n)$ is undefined.

Lambda Notation

We will often use Alonzo Church's *lambda notation*. Given a mathematical expression $a(x_1, \ldots, x_n)$ the function $f : \mathbb{N}_0^n \to \mathbb{N}_0$ that maps $(x_1, \ldots, x_n) \mapsto a(x_1, \ldots, x_n)$ may be denoted by $f = \lambda_{x_1,\ldots,x_n} \cdot a(x_1, \ldots, x_n)$.

1.2 The URM

Informal Discussion

An *algorithm* is a finite sequence of discrete mechanical instructions. A numerical function is *effectively computable* (or simply *computable*) if an algorithm exists that can be used to calculate the value of the function for any given input from its domain.

[1] Total functions and usual functions are equivalent.

The Unlimited Register Machine

The *unlimited register machine* has an infinite number of *registers* labelled $R_1, R_2, \ldots$, each containing a natural number, if R_i is a register then r_i is the number it contains. It can be represented as follows

R_1	R_2	R_3	R_4	R_5	R_6	R_7	$\cdots$
r_1	r_2	r_3	r_4	r_5	r_6	r_7	$\cdots$

The contents of the registers determine its *state* or *configuration*, which might be altered by the URM in response to certain *instructions*.

URM Programs

Name of Instruction	Instruction	URM response
Zero	`Z(`n`)`	$r_n \leftarrow 0$
Successor	`S(`n`)`	$r_n \leftarrow r_n + 1$
Transfer	`T(`m`,`n`)`	$r_n \leftarrow r_m$
Jump	`J(`m`,`n`,`q`)`	if $r_m = r_n$ then jump to q-th instruction; otherwise proceed to next instruction.

Without exception, the parameters of these instructions are elements of $\mathbb{N}_1$.

Definition 1.2. An *URM program* is a finite sequence of URM instructions. The number of instructions of a program is denoted by $\#P$.

Given a program $P = (I_1, \ldots, I_n)$ the URM always starts by executing I_1, the execution flow then proceeds incrementally unless a jump instruction is performed. The machine's response to each instruction is described in the table above.

Definition 1.3. An URM program P *computes* the function $f : \mathbb{N}_0^n \nrightarrow \mathbb{N}_0$ if for every $a_1, \ldots, a_n, b \in \mathbb{N}_0$ then:

$$P(a_1, \ldots, a_n) \downarrow b \Leftrightarrow (a_1, \ldots, a_n) \in \mathsf{Dom}\, f \wedge f(a_1, \ldots, a_n) = b,$$

and

$$P(a_1, \ldots, a_n) \uparrow \Leftrightarrow f(a_1, \ldots, a_n) = -$$

The class of URM-computable functions is denoted by $\mathcal{C}$ and by $\mathcal{C}_n$ the class of n-ary computable functions.

Definition 1.4. Let P be an URM program and $n \in \mathbb{N}_1$. The unique n-ary function that P computes is denoted by $f_P^{(n)}$, which, given any $x_1, \ldots, x_n \in \mathbb{N}_0$, is defined by:

$$f_P^{(n)}(x_1, \ldots, x_n)) = \begin{cases} - & \text{if } P(x_1, \ldots, x_n) \uparrow \\ y & \text{if } P(x_1, \ldots, x_n) \downarrow y \end{cases} .$$

Example 1.1. Let $Q = (\mathtt{Z(2)}, \mathtt{J(1,2,6)}, \mathtt{S(2)}, \mathtt{J(1,1,2)}, \mathtt{S(3)}, \mathtt{T(2,1)})$. The binary function computed by Q is $f_Q^{(2)}(x,y) = x$.

Definition 1.5. Let $M(x_1, \ldots, x_n)$ be an n-ary predicate. The characteristic function of the predicate M is the function $C_M : \mathbb{N}_0^n \to \mathbb{N}_0$ defined by:

$$C_M(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } M(x_1, \ldots, x_n) \\ 0 & \text{if } \neg M(x_1, \ldots, x_n) \end{cases} .$$

The predicate M is *decidable* if its characteristic function is computable and *undecidable* when it is not.

Building programs out of other programs

Definition 1.6. A program P is in *standard form* if, for every jump instruction $\mathtt{J(}m\mathtt{,}n\mathtt{,}q\mathtt{)} \in P$ it holds that $q \leq \#P + 1$.

Definition 1.7. Let $P = (I_1, \ldots, I_n)$ be an URM program. We denote by P^* the program $(I_1', \ldots, I_n')$ constructed as follows:

$$I_i' = \begin{cases} \mathtt{J(}m\mathtt{,}n\mathtt{,}\#P+1\mathtt{)} & \text{if } I_i = \mathtt{J(}m\mathtt{,}n\mathtt{,}k\mathtt{)},\ \text{with } k > \#P + 1 \\ I_i & \text{otherwise} \end{cases} .$$

Definition 1.8. Two programs P_1 and P_2 are *(strongly) equivalent* if, for any initial configuration $(a_1, a_2, a_3, \ldots)$ it holds that:

- $P_1(a_1, a_2, a_3, \ldots) \downarrow$ iff $P_2(a_1, a_2, a_3, \ldots) \downarrow$;
- when it is the case that both computations $P_1(a_1, a_2, a_3, \ldots)$ and $P_2(a_1, a_2, a_3, \ldots)$ stop, the final configurations of both machines are equal.

Theorem 1.1. *Let P be a URM program. The program P^* is in standard form and is equivalent to P.*

Definition 1.9. Let P and Q be URM programs. The *concatenation* of P and Q, denoted by $P;Q$, is the URM program defined as follows:

- $\#(P;Q) = \#P + \#Q$,
- For every $l \in \{1, \ldots, \#P\}$, $(P;Q)[l] = P'[l]$,
- For every $k \in \{1, \ldots, \#Q\}$:

$$(P;Q)[\#P + k] = \begin{cases} Q[k] & \text{if } Q[k] \text{ is not a jump instruction} \\ \mathtt{J(}m\mathtt{,}n\mathtt{,}r + \#P\mathtt{)} & \text{if } Q[k] = \mathtt{J(}m\mathtt{,}n\mathtt{,}r\mathtt{)} \end{cases} .$$

Definition 1.10. Let P be an URM program. If $\{R_{v_1}, \ldots, R_{v_n}\}$ is the set of registers mentioned in program P, we denote by $\rho(P)$ the number $\max\{v_1, \ldots, v_n\}$.

Definition 1.11. Let $n, j \geq 1$ and $i_1, \ldots, i_n > n$ and P be an URM program. We denote by $P[i_1, \ldots, i_n \to j]$ the following URM program:

$$(\mathtt{T}(i_1,1), \ldots, \mathtt{T}(i_n,n), \mathtt{Z}(n+1), \ldots, \mathtt{Z}(\rho(P))); P; (\mathtt{T}(1,j)),$$

where the sequence of instructions $\mathtt{Z}(n+1), \ldots, \mathtt{Z}(\rho(P))$ only occurs if $\rho(P) > n$.

Definition 1.12. A *generalized URM program* Q is a finite sequence of *generalized URM instructions* $(I_1, \ldots, I_k)$, with $k \geq 1$, where each each of this sequence's elements is either a standard URM instruction or one of the following two (called P-calling instructions):

1. `CallP`,
2. `CallP[`$i_1, \ldots, i_n \to j$`]`,

where $n, j \geq 1$ and $i_1, \ldots, i_n > n$ and P is an URM program which does not contain instructions that call program Q or instructions that call other programs that call program Q.

1.3 Primitive and Partial Recursive Functions

Definition 1.13. The following functions are called *basic functions*:

1. The *zero* functions: $\mathsf{zero} = \lambda_x \cdot 0$,
2. the *successor* function: $\mathsf{suc} = \lambda_x \cdot x + 1$,
3. for each $n \in \mathbb{N}_1$ and $i \in \{1, \ldots, n\}$, the *projection* function: $U_i^n = \lambda_{x_1,\ldots,x_n} \cdot x_i$.

Theorem 1.2. *The basic functions are URM-computable.*

Definition 1.14. Let $f : \mathbb{N}_0^k \nrightarrow \mathbb{N}_0$ and $g_i : \mathbb{N}_0^n \nrightarrow \mathbb{N}_0$, for each $i \in \{1, \ldots, k\}$. Define $g = (g_1, \ldots, g_k)$ as

$$\begin{aligned} g : \mathbb{N}_0^n &\nrightarrow \mathbb{N}_0^k \\ (x_1, \ldots, x_n) &\mapsto g(x_1, \ldots, x_n) \\ &\simeq (g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)). \end{aligned}$$

The composition of f and $g = (g_1, \ldots, g_k)$, denoted $f \circ g$, is the following function:

$$\begin{aligned} f \circ g : \mathbb{N}_0^n &\nrightarrow \mathbb{N}_0 \\ (x_1, \ldots, x_n) &\mapsto (f \circ g)(x_1, \ldots, x_n) \simeq f(g_(x_1, \ldots, x_n)) \\ &\simeq f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)). \end{aligned}$$

Both of these functions are defined at a point $(x_1, \ldots, x_n)$ if and only if $g_1, \ldots, g_n$ are defined at $(x_1, \ldots, x_n)$.

Theorem 1.3. *Let $f : \mathbb{N}_0^k \nrightarrow \mathbb{N}_0$ and $g_i : \mathbb{N}_0^n \nrightarrow \mathbb{N}_0$, for each $i \in \{1, \ldots, k\}$, are computable functions, then the function $h = f \circ (g_1, \ldots, g_k) : \mathbb{N}_0^n \nrightarrow \mathbb{N}0$ defined, for any $(x_1, \ldots, x_n) \in \mathbb{N}_0^n$, by $h(x_1, \ldots, x_n) \simeq f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$, is also computable. In other words, composition preserves computability.*

Definition 1.15. Let $k \in \mathbb{N}_0$ and $n \in \mathbb{N}_1$. We denote by $\mathsf{k}^{(n)}$ the n-ary constant function define as follows:

$$\begin{aligned} \mathsf{k}^{(n)} &: \mathbb{N}_0^n \nrightarrow \mathbb{N}_0 \\ (x_1, \ldots, x_n) &\mapsto \mathsf{k}^{(n)}(x_1, \ldots, x_n) = k \end{aligned}$$

Theorem 1.4. *For every $k \in \mathbb{N}_0$ and every $n \in \mathbb{N}_1$ the function $\mathsf{k}^{(n)}$ is obtained by composition of basic functions.*

1.4 Bounded and Unbounded Quantification

1.5 Alternative Models of Computability

Turing Machines

A *Turing Machine* M is an abstract device which "performs" operations on a tape of infinite length in both directions. This tape is divided in individual squares along its length. At any given moment each square contains a single symbol from a fixed and finite set called the *alphabet of* $M = \{s_0, \ldots, s_n\}$. We assume that s_0 is the *blank* symbol β used to denote an empty square. The machine M has a *reading head* which, at any given moment, scans and reads a single square of the tape. This machine is capable of performing the following three kinds of operations on the tape:

1. Erase the symbol of the square being scanned and write one of the symbols in the alphabet;

2. Move the reading head one square to the right of the square being scanned;

3. Move the reading head one square to the left of the square being scanned.

At any given moment, the machine M is in one of a finite number of *states*, represented by $q_1, q_2, \ldots, q_m$. The execution of an operation may cause the state of M to change.

The operation to be performed by M is determined by its *specification*, denoted by Q, and its current state. The set Q is finite and its elements are quadruples, each of which is of the form (q_i, s_j, s_k, q_l), or (q_i, s_j, R, q_l), or (q_i, s_l, L, q_l), with $i, l \in \{1, \ldots, m\}$ and $j, k \in \{1, \ldots, m\}$.

A quadruple of the form (q_i, s_j, α, q_l) (where $\alpha \in \{s_k, R, L\}$ in Q specifies the action to be perform by M when it is in state q_i and reading the symbol s_j, as follows:

1. Execute the following operation on the tape:

   If $\alpha = s_k$, erase s_j and write s_k in the square being scanned;

   If $\alpha = R$, move the reading head one square to the right;

   If $\alpha = L$, move the reading head one square to the left.

2. Change into state q_l.

In order for a machine M to begin a computation an initial state is required and its reading head must be positioned over a single square of a given tape. Then M starts performing the actions determined by its specification, as described above. The computation terminates if M is in a state q_p and reading a symbol s_r such that there is no quadruple in Q which starts with $q_p\, s_r$. Unless stated otherwise, given a Turing machine M and an infinite tape, M starts its computation in the state q_1 and with its reading head placed over the leftmost non-blank square of the received tape.

Example 1.2. Let M be a Turing machine with alphabet $\{\beta, a, b\}$. This machine may be in only one of two states q_1 and q_2. Let Q be the following specification of M:

$$\begin{gathered} q_1\, a\, b\, q_2 \\ q_1\, b\, a\, q_2 \\ q_2\, a\, R\, q_1 \\ q_2\, b\, R\, q_1 \end{gathered}$$

A numerical function $f : \mathbb{N}_0^n \nrightarrow \mathbb{N}_0$ is *Turing-computable* if there exists a Turing machine that computes f. The class such functions is denoted by $\mathcal{T}$.

2 Formal Methods

2.1 Hoare Logic

Hoare logic is a formal system appropriate for proving the partial correctness of imperative programs.

2.2 Recursive Algorithms and Correctness

Definition 2.1. A *well-founded* relation on a set A is a binary relation R which does not contain infinite descending chains, i.e. there is no infinite sequence $a_0, a_1, \ldots$ of elements of A such that for every $n \in \mathbb{N}_0$ it holds that $a_{n+1} R a_n$.

The symbol "$\prec$" will often be used to denote a well-founded relation on a set A. Suppose $x, y \in A$, if $x \prec y$ then x is said to be a *predecessor* of y.

Theorem 2.1. *Let $\prec$ be a binary relation on a set A. Then $\prec$ is a well-founded relation on A if and only if every non-empty subset X of A contains a minimal element, i.e., there exists some c such that: $c \in X$ and $\nexists_a \in x : a \prec c$.*

In all that follows L denotes the set of of all lists (of any kind of elements) and L^+ denotes the set of all non-empty lists. Furthermore, given a list w:

- $\#w$ denotes the length w;
- w_i (with $w \neq \{\}$ and $i \in \{1, \dots, \#w\}$) denotes the element which is in the position i of list w;
- $w \setminus w_i$ (with $w \neq \{\}$ and $i \in \{1, \dots, \#w\}$) denotes the list that is obtained from w by deleting the element contained in position i.

Definition of functions using recursion

Theorem 2.2. *There exists one and only one function $u : A \to B$ such that:*

1. *For any minimal element y of A:*
$$u(y) = f(y),$$
for some fixed function $f : A \to B$, i.e. the value $u(y)$ is defined explicitly in terms of y without using the value of u at any other point.

2. *For any non-minimal element y of A*
$$u(y) = g(y, \{(k, u(k)) : k \prec y\},$$
for some fixed function g which is defined at every pair of the form (y, R_y), where y is a non-minimal element of A, and $R_y = \{(a, b) \in A \times B : a \prec y\}$ and is a functional relation.

A unuary function $u : A \to B$ is said to be *well defined recursively* if it satisfied both properties of theorem 2.2.

Theorem 2.3. *Let $n \in \mathbb{N}_1$ and let $X_1, \dots, X_n$ be any sets. There exists one and only one function $u : X_1 \times \cdots \times X_n \times A \to B$ such that:*

1. *For any $x_i \in A$ (with $i \in \{1, \dots, n\}$) and any minimal element y of A*
$$u(x_1, \dots, x_n, y) = f(x_1, \dots, x_n, y),$$
for some fixed function $f : X_1 \times \cdots X_n \times A \to B$.

2. *For $x_i \in X_i$ (with $i \in \{1, \dots, n\}$) and any non-minial element y of A*
$$u(x_1, \dots, x_n, y) = g(x_1, \dots, x_n, y, \{(k, u(x_1, \dots, x_n, k)) : k \prec y\}),$$
for some fixed function g which is defined at every n+2-uples of the form $(x_1, ldots, x_n, y, R_y)$, where $x_i \in X_i$, y is a non-minimal element of A, and $R_y = \{(a, b) \in A \times B : a \prec y\}$ and is a functional relation.

With theorem 2.3 we generalize the notion of recursion to $n + 1$-ary functions.

2.3 Algebraic Specification

Definition 2.2. A *signature* is a triple $\Sigma = (S, \mathsf{Op}, \mathsf{type})$, where:

- S is a non-empty set, whose elements are called *sorts*;
- Op is a non-empty set, whose elements are called *(symbols of) operations*, such that $S \cap \mathsf{Op} = \emptyset$;
- $\mathsf{type} : \mathsf{Op} \to S^* \times S$, with $S^* = \{\varepsilon\} \cup \{s_1 \ldots s_n : n \in \mathbb{N}_1 \text{ and } \forall_{i=1}^{n} s_i \in S\}$, where ε denotes the empty sequence; is a function that associates to each operation:

  Its arity (i.e. number of arguments) and the sequence that indicates the sort of each one of its arguments;

  The sort of its outcome.

  This information is designated by *declaration* or *sort* or *type* of an operation.

In the specification of abstract data types constants are usually presented as 0-ary operations.

Let $\circ \in \mathsf{Op}$ be an operation such that $\mathsf{type}(\circ) = (s_1 \ldots s_n, s)$, with $n \in \mathbb{N}_1$. The sequence $s_1 \ldots s_n$ indicates the type of each one of the n arguments of $\circ$ and s is the type of its outcome.

If $\circ \in \mathsf{Op}$ is an operation such that $\mathsf{type}(\circ) = (\varepsilon, s)$ then $\circ$ is a constant. In this case we say that $\circ$ is a constant of type s.

In what follows we shall often write $\circ : s_1 \ldots s_n \to s$ to indicate that $\circ \in \mathsf{Op}$ and $\mathsf{type}(\circ) = (s_1 \ldots s_n, s)$, with $n \in \mathbb{N}_1$. Analogously, we shall write $\circ :\to s$ to indicate that $\circ \in \mathsf{Op}$ and $\mathsf{type}(\circ) = (\varepsilon, s)$, i.e. $\circ :\to s$ means that $\circ$ is a constant of type s.