Theory of Computation

S. Arun-Kumar

February 18, 2009

Preface

Students of computer science in IIT Delhi usually take a course on Theory of Computation as an elective some time after they have finished courses on programming, data structures, discrete mathematical structures (whatever that means), computer architecture, programming languages and sometimes operating systems as well.

Theory of computation is of course a very broad and deep area, and it is anyone's guess what really should be taught in such course. For example, Dexter Kozen's text with the same name suggests that the course should dwell primarily on complexity classes. Most courses on Theory of Computation in India follow the classic text by Hopcroft and Ullman [1] on formal languages and automata which was written at a time when parsing, compiling, code optimization and complexity theory were pursued as frontier areas of research. As a result, the exposure to automata theory and formal languages was considered the most important aspect of the theory, later followed by some exposure to NP-completeness. Moreover the book was written for graduate students [1].

More recently (apparently on the strength of having course web pages on Programming, Logic, Programming Languages and Complier Design), I have been bombarded by questions from students and outsiders (some of them managers with computer science degrees who are willing to pay thousands of dollars if I agreed to sit in on such a project) regarding the feasibility of designing tools which would take the programs of their (novice?) programmers and transform them automatically into the most efficient program possible or the shortest program possible in some language. I have even been asked why these programs cannot be proved automatically thus saving the company a small fortune in testing time and being able to meet stiff deadlines.

Anyway that's my story and I am sticking to it.

It is my feeling that at a time when courses on compilers, formal languages, complexity theory and automatic verification are usually electives which students tend to avoid, this has led to a disconnect between a course primarily devoted to formal languages and automata theory and the bouquet of courses that students otherwise need to do in a typical computer science curriculum.

Anyway that's my story and I am sticking to it.

It is also my feeling that IIT students (who are numerically quite literate) feel most comfortable when numbers and computations on numbers form the basis for a theory of computation. On the other hand courses on theory of computation which primarily teach automata and formal languages usually completely ignore the connections between programming and computability theory and scant attention is paid to the theory of primitive recursive functions and the design of data structures or programming language features. As a result students who sit through a course on automata and formal languages and who have no intention of pursuing either

[1]It is a mystery to me how it has come to be taught as a textbook for an *undergraduate* course. I would assume that the same natural processes which brought Euclid, Descartes and Newton down to school mathematics are responsible for it. But nevertheless the transition in the case of formal languages and automata has taken place in less than 15 years while Newton took about 200 years before being prescribed for high school.

compilers or formal verification or complexity theory, emerge from the course feeling that the whole course was useless and unconnected to anything they have learned and anything else they might learn and of no use whatsoever in their professional life in industry.

Anyway that's my story and I am sticking to it.

I decided therefore to make computability theory the primary focus of these lecture notes and gradually introduce Turing machines, finite automata and formal languages. I have used the books of Cutland [4] and Martin Davis ([2], [3]) as my primary sources for these lecture notes. I have tried to introduce the connections between the theory of computability with other courses such as programming, functional programming, data structures, discrete mathematical strucutres and operating systems in as elementary a fashion as possible. It has been a source of wonder to me that this beautiful material seems to be taught mostly only to students of pure or applied mathematics or logic rather than to computer science students. I am sure students of engineering can absorb this material and appreciate it equally well, especially since it requires little more than high school mathematics to understand it (though it might require a more maturity to appreciate it).

Anyway that's my story and I am sticking to it.

I have of course borrowed freely from various sources, but in many cases I have made significant departures fromt he original material especially as concerns notation. Not to put too fine a point on it, I believe that my notation though strange to look at, is an attempt to clean up (mostly for my own understanding) misleading notation used in various sources.

Contents

1 Basics: Prerequisites and Notation **7**

1.1 Sets and Operations on Sets 7

1.2 Relations, Functions and Predicates 8

1.3 Equality 13

1.4 Other Notational Devices 15

2 The RAM Model **17**

2.1 The Unlimited Register Machine (URM) 17

2.2 Examples 21

3 The Primitive Recursive Functions **29**

3.1 Introduction 29

3.2 Primitive Recursion 31

3.3 Primitive Recursive Predicates 39

3.4 Some Forms of Primitive Recursion 40

4 The Partial Recursive Functions **45**

5 Coding and Gödelization **49**

5.1 Introduction 49

5.2 Numbering and Coding 49

5.3 Gödel Numbers of URM Instructions 51

5.4 Gödel Numbering of Programs 51

6 The Hierarchy of Primitive Recursive Functions 53

6.1 Introduction 53

6.2 URM–: A Restricted Machine for Primitive Recursion 54

6.3 A Non-Primitive Recursive Total Partial Recursive function 56

6.3.1 A Variant of the Ackermann function 56

6.3.2 The Growth of functions 59

7 Universality and Parametrisation 65

7.1 The Parametrisation theorem (Simplified form) 66

7.2 Universal functions and Programs 68

8 The type-free λ-calculus 77

Chapter 1

Basics: Prerequisites and Notation

1.1 Sets and Operations on Sets

We assume the usual notions and notations of set theory including those of relations.

We will generally use the following notation for sets.

$\mathbb{N}$	The set of *Natural numbers* (including 0)
$\mathbb{P}$	The set of *Positive integers*
$\mathbb{R}$	The set of *Real numbers*
$\mathbb{Z}$	The set of *Integers*
$\emptyset$	The *empty* set
$\subseteq$	The (infix) *subset* relation between sets
$\subset$	The (infix) *proper subset* relation between sets
$\cup$	The infix *union* operation on sets
$\cap$	The infix *intersection* operation on sets
$\sim$	The prefix *complementation* operation on sets
$-$	The infix *set difference* operation on sets
$\times$	The infix *cartesian product* of sets
A^n	The postfix *n-fold cartesian product* of A, i.e.$\underbrace{A \times \cdots \times A}_{n\text{-times}}$
$\mathbf{2}^A$	The *powerset* of A

The usual notations will be used for arithmetic operations on numbers. This includes the operation of subtraction (denoted by the infix operator $-$) which is also the set difference operation. Similarly we may use the symbol $\times$ for both cartesian product and multiplication of numbers. Usually the context will determine which operation is intended. Some operations such as multiplication may be denoted by ., or no symbol at all as is usual in mathematics. Further the usual prefix operations for sum ($\sum$) and product ($\prod$) of sets or sequences of numbers will be used.

Convention 1.1.1

- *A^0 will be identified with the empty set $\emptyset$.*
- *A^1 will be identiifed with the set A*
- *A 1-tuple (a) will be identified with a.*

1.2 Relations, Functions and Predicates

Definition 1.1 *A n-ary* **relation** *for $n > 0$ on sets $A_1, \ldots, A_n$ is any subset of $A_1 \times \cdots \times A_n$.*

Definition 1.2 *Given two sets A and B and a binary relation $f \subseteq A \times B$,*

- *Then $f^{-1} \subseteq B \times A$ the* **converse** *of f is the relation defined by $(b, a) \in f^{-1}$ if and only if $(a, b) \in f$.*
- *f is called a* **partial function** *of type $A \rightarrow\!\!\!\!\!+ B$, if f satisfies the* **image uniqueness** *property viz. that for each element $a \in A$ there is* at most one *element $b \in B$ such that $(a, b) \in f$. b is called the* **image** *of a under f.*
- *f is said to be* **defined** *at $a \in A$ if a has an image under f.*
- *f is said to be* **undefined** *at a if it has no image under f.*
- *f is called a* **total function** *of type $A \rightarrow B$ if f is a partial function of type $A \rightarrow\!\!\!\!\!+ B$ and every element of A has an image under f.*

$r \subseteq A_1 \times \cdots \times A_n$	A n-ary (for $n > 0$) relation on sets $A_1, \ldots, A_n$
$f : A \rightarrow\!\!\!\!\!+ B$	A *partial* function from set A to set B
$f : A \rightarrow B$	A *total* function from set A to set B
$f(a) = b$	b is the **image** of a under f
$f(a) = \perp$	f is **undefined** at a
$\mathsf{Dom}(f)$	The subset of values for which f is *defined*
$\mathsf{Ran}(f)$	The subset of values which are images of elements in $\mathsf{Dom}(f)$

Notes.

1. A **constant** (e.g. a natural number) will be regarded as a total function from the empty set to the set that constant belongs to (e.g. $\mathbb{N}$). This is to be regarded as being different from a **constant function** such as $f : \mathbb{R} \rightarrow \mathbb{N}$ defined as $f(x) = 5$ which is a unary total function defined on the reals and which yields the natural number 5 for every real argument.

2. The notation we use for functions may also be used for relations. In fact every relation $r \subseteq A_1 \times \cdots \times A_n$ may be thought of as a *total* function of type $r : A_1 \times \cdots \times A_k \rightarrow \mathbf{2}^{A_{k+1} \times \cdots \times A_n}$, for each k with $1 \leq k < n$. Hence for any $(a_1, \ldots, a_k) \in A_1 \times \cdots \times A_k$, where $1 \leq k < n$, then

$$r(a_1, \ldots, a_k) = \{(a_{k+1}, \ldots, a_n) \mid (a_1, \ldots, a_k, a_{k+1}, \ldots, a_n) \in r\} \tag{1.1}$$

3. When $r(a_1, \ldots, a_k)$ is a singleton set we omit the set braces. Further, each such relation may also be thought of as a *partial* function of type

$$r : A_1 \times \cdots \times A_k \not\longrightarrow A_{k+1} \times \cdots \times A_n \tag{1.2}$$

where $(a_1, \ldots, a_k) \in \mathsf{Dom}(r)$ if and only if $r(a_1, \ldots, a_k) \neq \emptyset$ in 1.1

Facts 1.1

1. *Every total function is also a partial function.*
2. $f : A \rightarrow B$ *implies* $\mathsf{Dom}(f) = A$.

Definition 1.3 *Two partial functions* $f, g : A \not\longrightarrow B$ *are* **equal** *if and only if* $\mathsf{Dom}(f) = \mathsf{Dom}(g)$ *and for each* $a \in \mathsf{Dom}(f)$, $f(a) = g(a)$.

Definition 1.4 *Let* $f : A \not\longrightarrow B$ *be a partial function.*

- f *is* **injective** *if for each* $a, a' \in \mathsf{Dom}(f)$, $a \neq a'$ *implies* $f(a) \neq f(a')$. f *is also said to be an* **injection**.
- *If* f *is injective then* $f^{-1} : B \not\longrightarrow A$ *the* **inverse of** f, *with* $\mathsf{Dom}(f^{-1}) = \mathsf{Ran}(f) \subseteq B$ *and for every* $b \in \mathsf{Dom}(f^{-1})$, $f^{-1}(b) = a$ *iff* $f(a) = b$.
- f *is* **surjective** *if* $\mathsf{Ran}(f) = B$. f *is also said to be a* **surjection**.
- f *is* **bijective** *(or is a* **bijection***) if it is both injective and surjective.*

Facts 1.2

1. *If $f : A \not\rightarrow B$ is injective then for every $a \in \mathsf{Dom}(f)$, $f^{-1}(f(a)) = a$.*
2. *If $f : A \rightarrowtail\!\!\mapsto B$ is bijective then f^{-1} is total (of type $B \rightarrow A$) and for every $b \in B$, $f(f^{-1}(b)) = b$.*

Definition 1.5 *Let $g : A \not\rightarrow B$ and $f : B \not\rightarrow C$ be partial functions. The* **composition** *of g and f (denoted $[g; f]$ or $[f \circ g]$) is the partial function $h : A \not\rightarrow C$ such that*

- $\mathsf{Dom}(h) = \{a \in \mathsf{Dom}(g) \mid g(a) \in \mathsf{Dom}(f)\}$ *and*
- *for each $a \in \mathsf{Dom}(h)$, $h(a) = [g; f](a) = [f \circ g](a) = f(g(a))$.*

Notation. The notation $f \circ g$ is preferred by mathematicians generally, though the notation $g; f$ more closely represents the sequential composition of functions in programming. We will use either of them.

Note that composition is **strict** in the sense that for any a, $a \notin \mathsf{Dom}(g)$ or $a \in \mathsf{Dom}(g)$ but $g(a) \notin \mathsf{Dom}(f)$ implies $a \notin \mathsf{Dom}(h)$ and hence $h(a) =\perp$.

If we were to regard f and g as transducers which take an input from its domain and transform it into an output in its co-domain, then h is a composite transducer which transforms an input from $a \in A$ into an output $c \in C$ in a two step process by first transforming a to $b = g(a)$ and then transforming b to $c = f(b) = f(g(a))$. The two step transformation may be pictured as follows.

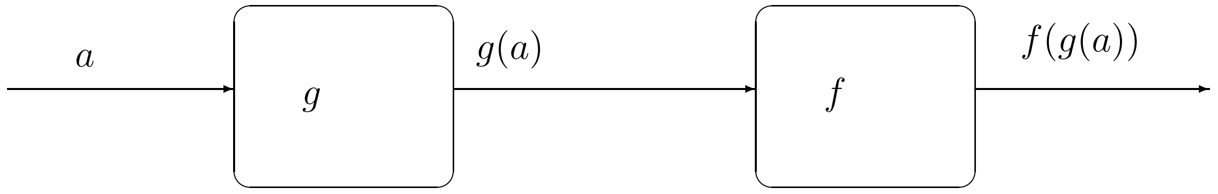


Figure 1.1: Composition of unary functions

Notation. We usually abbreviate sequences of values such as $a_1, \ldots, a_k$ (which are the arguments of a function) by $\overrightarrow{a}$, where the length of the sequence is either understood or immaterial. Where the length of the sequence is important we may write it as $\overrightarrow{a}^k$.

We will be interested in a generalised composition as defined below.

Definition 1.6 *Assume $f : B^m \not\rightarrow C$ is an m-ary function and $g_1, \cdots, g_m$ are k-ary functions of the type $g_1, \cdots, g_m : A^k \not\rightarrow B$. Then the k-ary function $h : A^k \not\rightarrow C$ obtained by* **composing** *the m functions $g_1, \ldots, g_m$ with f (as shown in the figure below) is such that*

- $\mathsf{Dom}(h) = \{\overrightarrow{a}^k \in A^k \mid (\forall i : 1 \leq i \leq m : \overrightarrow{a}^k \in \mathsf{Dom}(g_i)) \wedge (g_1(\overrightarrow{a}^k), \cdots, g_m(\overrightarrow{a}^k)) \in \mathsf{Dom}(f)\}$ *and*
- $h(\overrightarrow{a}^k) = f(g_1(\overrightarrow{a}^k), \cdots, g_m(\overrightarrow{a}^k))$.

Notation. The same notation for sequences of values may be extended to sequences of functions which are themselves arguments of a higher-order function. For example, the tuple of functions $g_1, \cdots, g_m$ (which are arguments of the generalised composition operator) may be denoted $\overrightarrow{g}$. Extending the notation for the composition of unary functions to generalised composition we have $h = \overrightarrow{g}; f = f \circ \overrightarrow{g}$. Where the length of the tuple is important we may write the above tuples as $\overrightarrow{a}^k$ and $\overrightarrow{g}^m$ respectively. Then $h = \overrightarrow{g}^m; f = f \circ \overrightarrow{g}^m$.

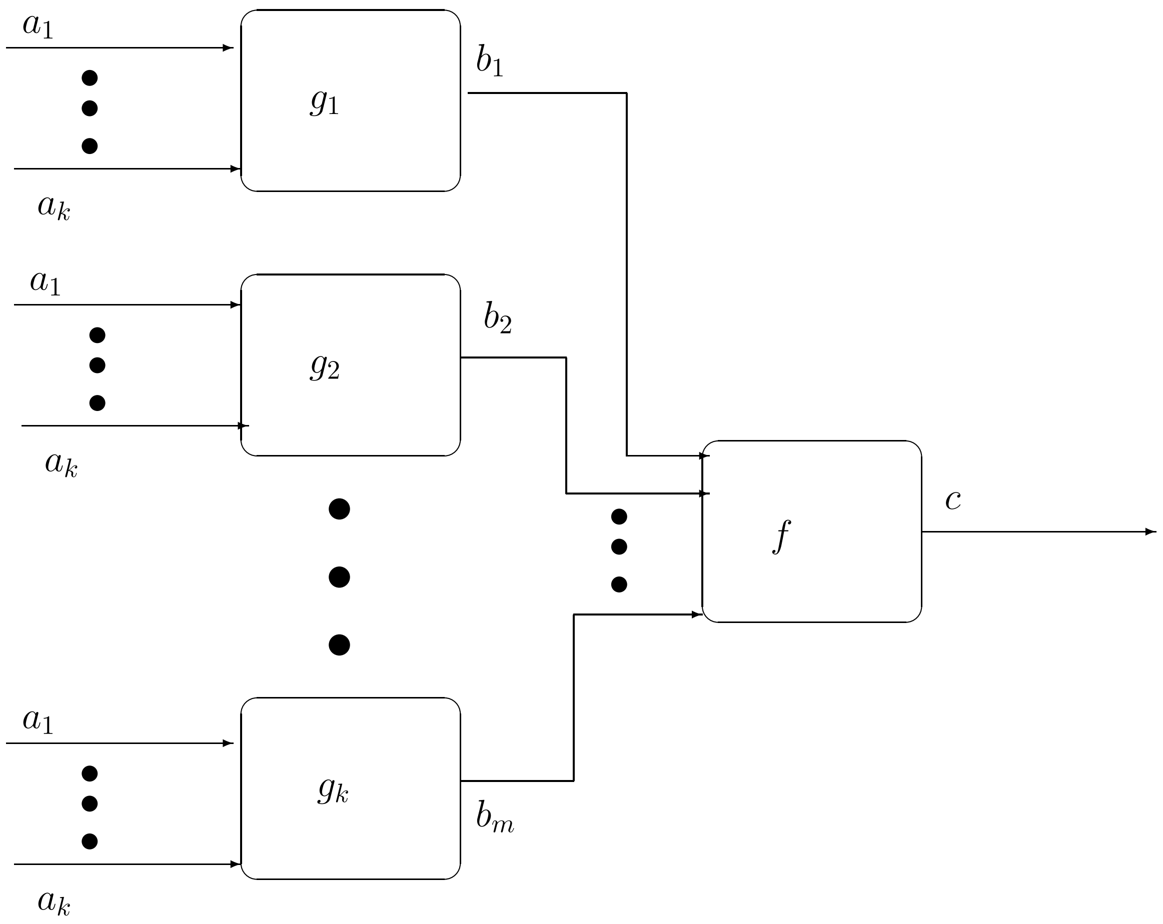


Figure 1.2: Generalised Composition of functions

Definition 1.7 *Given an n-ary relation $r \subseteq A_1 \times \cdots \times A_n$, its* **characteristic function** *denoted χ_r, is the* total *function $\chi_r : A_1 \times \cdots \times A_n \to \{0, 1\}$ such that for each $\overrightarrow{a} \in A_1 \times \cdots \times A_n$,*

$$\chi_r(\overrightarrow{a}) = \begin{cases} 1 & \text{if } \overrightarrow{a} \in r \\ 0 & \text{if } \overrightarrow{a} \notin r \end{cases}$$

Definition 1.8 *An* **atomic predicate** *is a relation*[1]*. The class of* **(first-order) predicates** *is the smallest class containing the atomic predicates and closed under the following* **logical** *operations.*

1. *the unary (prefix)* negation *operation (denoted $\neg$)*
2. *the binary (infix) operation* and *(denoted $\wedge$)*
3. *the binary (infix) operation* or *(denoted $\vee$)*
4. *the unary (prefix)* universal quantifier *(denoted $\forall$)*
5. *the unary (prefix)* existential quantifier *(denoted $\exists$)*

The usual notions of free and bound variables apply to the predicates and quantified formulae. Further the usual meaning of these predicates is as defined in any text on first order logic. Further the usual rules of **scope** of variables apply here. To denote the scope of variables in quantified predicates we write the body of the quantified formula within brackets.

Example 1.2.1 *Consider the first order predicate $q(\overrightarrow{y})$ defined as $q(\overrightarrow{y}) = \forall \overrightarrow{x}\ [p(\overrightarrow{x}, \overrightarrow{y})]$ where $p(\overrightarrow{x}, \overrightarrow{y})$ is a first order predicate in which the variables in $\overrightarrow{x}$ and $\overrightarrow{y}$ may occur free. $p(\overrightarrow{x}, \overrightarrow{y})$ may itself be a compound predicate defined in terms of other predicates. The quantifier "$\forall \overrightarrow{x}$" in $q(\overrightarrow{y})$ however,* binds *all occurrences of variables in $\overrightarrow{x}$ that occur* free *in $p(\overrightarrow{x}, \overrightarrow{y})$. The scope of the binding is indicated by the matching brackets "$[$" and "$]$" in the definition of $q(\overrightarrow{y})$. The variables that are* free *in q are precisely those from $\overrightarrow{y}$ whixh occur free in $p(\overrightarrow{x}, \overrightarrow{y})$ and this fact is indicated by the notation "$q(\overrightarrow{y})$".*

Definition 1.9 *Let p and q be predicates with free variables drawn from $\overrightarrow{x}$ and let $\overrightarrow{x} = y, \overrightarrow{z}$. The characteristic functions of compound predicates is defined in terms of the characteristic functions of their components as follows.*

- $\chi_{\neg p}(\overrightarrow{x}) = 1 - \chi_p(\overrightarrow{x})$
- $\chi_{p \wedge q}(\overrightarrow{x}) = \chi_p(\overrightarrow{x}).\chi_q(\overrightarrow{x})$
- $\chi_{p \vee q}(\overrightarrow{x}) = 1 - (\chi_{\neg p}(\overrightarrow{x}).\chi_{\neg q}(\overrightarrow{x}))$
- $\chi_{\forall y[p]}(\overrightarrow{z}) = \begin{cases} 0 & \textit{if } \chi_p(a, \overrightarrow{z}) = 0 \textit{ for some } a \\ 1 & \textit{otherwise} \end{cases}$

[1]Hence we use the same notation for atomic predicates as we use for relations

- $\chi_{\forall y[p]}(\overrightarrow{z}) = \begin{cases} 1 & \textit{if } \chi_p(a, \overrightarrow{z}) = 1 \textit{ for some } a \\ 0 & \textit{otherwise} \end{cases}$

We use some derived operators such as $\Rightarrow$ and $\Leftrightarrow$ defined as follows:

$$\begin{array}{rcl} (p \Rightarrow q) & = & (\neg p) \vee q \\ (p \Leftrightarrow q) & = & (p \Rightarrow q) \wedge (q \Rightarrow p) \end{array}$$

The usual equational laws of First-order logic hold. We will also have occasion to use **bounded quantification** – operations derived from the quantifiers and used in the context of the naturals. They are defined as follows.

$$\begin{array}{rcl} \forall y < z[p(\overrightarrow{x}, y)] & = & \forall y[(0 \leq y < z) \Rightarrow p(\overrightarrow{x}, y)] \\ \exists y < z[p(\overrightarrow{x}, y)] & = & \exists y[(0 \leq y < z) \wedge p(\overrightarrow{x}, y)] \end{array}$$

Note that bounded quantification obeys the De Morgan laws making each bounded quantifier the dual of the other. That is

$$\begin{array}{rcl} \forall y < z[p(\overrightarrow{x}, y)] & = & \neg(\exists y < z[\neg p(\overrightarrow{x}, y)]) \\ \exists y < z[p(\overrightarrow{x}, y)] & = & \neg(\forall y < z[\neg p(\overrightarrow{x}, y)]) \end{array}$$

1.3 Equality

In the course of these notes we shall use various kinds of equality. Most of these notions of equality share the property of being congruences i.e. they are equivalence (reflexive, symmetric and transitive) relations and the two sides of the equality are mutually replaceable under all contexts. However, it is necessary to emphasize that there are differences between the various kinds of equality. Most mathematics texts often ignore these differences, (and we may have been guilty too in the course of writing these notes!) but we feel that they are important for various logical reasons and the reader should be aware of the differences. We briefly describe them below and we hope the differences that we have made explicit will be useful for the reader in understanding not just these notes but mathematics texts in general.

Definitional Equality. This is an equality by definition, and we use the symbol $\triangleq$ to denote this. This is most often used in the following situations.

1. to abbreviate a complex expression by giving it a name,
2. to name a complex expression because it or its form is somehow considered important.

Hence when we write $p(x) \triangleq x^2 - 9$ or $q(x) \triangleq x^3 + 3x^2 + 3x + 1$, we are either abbreviating or naming the expression on the right hand side by the expression on the left hand side of $\triangleq$. Given another definition $r(x) \triangleq (x+1)^3$, even in the context of polynomials over the reals or integers it is incorrect to write "$q(x) \triangleq r(x)$" because the two expressions $q(x)$ and $r(x)$ define *syntactically* distinct expressions.

Instantiated Equality. Continuing with the previous examples we would often write expressions like $p(4) = 4^2 - 9 = 7$. Note here that $p(4)$ is an *instance* of $p(x)$ defined above. The equality symbol "=" (between "$p(4)$" and the expression "$4^2 - 9$") here is overloaded to imply that $p(4)$ is a particluar instance of the definition $p(x) \triangleq x^2 - 9$[2] Note that "$p(y) = y^2 - 9$" is also a form of instantiated equality and so is $p(x+1) = (x+1)^2 - 9$.

Syntactic Identity This is a peculiar notion of equality we will use more in the context of languages or programs rather than in the case of mathematical expressions. Consider the two expressions $r(x) \triangleq (x+1)^3$ and $s(y) \triangleq y^3$. It is clear that by appropriate instantiations, we get $r(z) = (z+1)^3$ and $s(z+1) = (z+1)^3$. Therefore $r(z) = s(z+1)$. However, this equality between $r(z)$ and $s(z+1)$ is stronger than just provable equality — they are in fact, *syntactically identical* as expressions. This syntactic identity depends only on the form of the original definitions and is independent of the context or the interpretation of these expressions in any mathematical domain. We emphasize this aspect by writing $r(z) \equiv s(z+1)$. As in the case of definitional equality, syntactic identity also implies provably equality. Also definitional equality implies syntactic identity.

α-Equality. Consider the two definitions $p(x) \triangleq x^2 - 9$ and $p'(y) \triangleq y^2 - 9$. It is clear that $p(y)$ may be regarded as an instance of the definition $p(x) \triangleq x^2 - 9$ and $p(y) \equiv p'(y)$. Equally well may $p'(x)$ be regarded as an instance of the definition $p'(y) \triangleq y^2 - 9$ yielding $p(x) \equiv p'(x)$. However the two definitions themselves are not syntactically identical because of the use of different[3] names x and y to express each definition. Hence $p(x) \not\equiv p'(y)$. However there is a sense in which both the definitions $p'(y) \triangleq y^2 - 9$ and $p(x) \triangleq x^2 - 9$ may be regarded as being the same (or equal) in that the variables x and y used to give the definition its form are only "place-holders" whose name is of no consequence whatsoever except when replacement of one name by the other causes a name clash. In the absence of such name clashes the uniform replacement of one name by the other causes neither any confusion nor any change in the intended meaning of the definition. We call this *α-equality* and we denote this fact by writing $p(x) =_\alpha p'(y)$. As with other equalities, since α-equality is also a reflexive relation, it follows that any two syntactically identical expressions are also α-equal. We will come across this later in the λ-calculus.

Provable Equality. This is the most commonly used (and abused) equality and is denoted by the usual "=" symbol. In the foregoing examples even though it is incorrect to write $q(x) \triangleq r(x)$, in the context of polynomials over real numbers it is perfectly correct to write $(x+1)^3 = x^3 + 3x^2 + 3x + 1$, because the equality of the expressions $x^3 + 3x^2 + 3x + 1$ and $(x+1)^3$ *can be proven* using the laws of real number arithmetic. Any two expressions that are equal by definition are also provably equal (by invoking the *definition*). Hence

[2]However the equality symbol "=" between the expressions "$4^2 - 9$" and "7" is provable equality.
[3]The fact that we have used two different names "p" and "p'" does not count!

$p(x) = x^2 - 9$ and we could replace all occurrences of the expression $p(x)$ by the expression $x^2 - 9$ and vice-versa. It is also perfectly correct to write "$p(x) = x^2 - 9$" or "$x^2 - 9 = p(x)$". To continue with our examples in the context of polynomials, it follows therefore that $q(x) = r(x)$ since the expressions they denote are *provably equal.* But note that even in the context of polynomials over reals, $q(x) \neq_\alpha r(x)$ and $q(x) \not\equiv r(x)$.

To summarize, we have that for any two (mathematical or formal) expressions E and F

1. $E \triangleq F$ implies $E \equiv F$ and
2. $E \equiv F$ implies $E =_\alpha F$ which in turn implies $E = F$. Hence $E \triangleq F$ also implies $E =_\alpha F$ and $E = F$.
3. None of the above implications may however, be reversed in general i.e., it is not true in general that if $E = F$ then $E =_\alpha F$ or $E \equiv F$ would hold, nor does $E \equiv F$ imply $E \triangleq F$ or $F \triangleq E$ holds.
4. Further while all forms of equality may be instantiated, no form of instantiated equality may imply any other form of equality unless it can be proven.

1.4 Other Notational Devices

We generally follow the practice of splitting long proofs of theorems into claims and using these claims. In most cases the proofs of claims are distinguished from the main proof by the use of the bracketing symbols "$\vdash$" and "$\dashv$" to enclose the proof of the claim and separate it from the proof of the main theorem.

Chapter 2

The RAM Model

2.1 The Unlimited Register Machine (URM)

Assume the following about the machine.

- A countably infinite number of discrete storage elements called **registers** each of which can store an arbitrarily large natural number. The registers $R_1, R_2, R_3, \cdots$ are indexed by the set $\mathbb{P}$ of positive integers. These registers form the **memory**.
- The contents of R_i, $i \in \mathbb{P}$ are denoted respectively by $!R_i \in \mathbb{N}$. Equivalently $M : \mathbb{P} \to \mathbb{N}$ the **memory map** may be regarded as a total function from the positive integers to the natural numbers so that $M(i) = !R_i$ for each $i \in \mathbb{P}$.
- We will usually ensure that *only finitely many* registers contain *non-zero values* i.e. M is 0 $\underline{\text{almost everywhere}}$.
- A URM program P is a sequence $I_1, I_2, \ldots, I_p$ of **instructions**.
- The instruction set has opcodes $0, 1, 2, 3$ respectively, and each instruction I_j, $1 \leq j \leq p$, is typically of one of the following forms where $i, m, n \in \mathbb{P}$.

opcode	instruction	semantics	Verbal description
0	$\mathsf{Z}(n)$	$R_n := 0$	Clear register R_n
1	$\mathsf{S}(n)$	$R_n := !R_n + 1$	Increment the contents of register R_n
2	$\mathsf{C}(m, n)$	$R_n := !R_m$	Copy the contents of register R_m into R_n
3	$\mathsf{J}(m, n, i)$	$!R_m = !R_n \ ? \ i : j + 1$	Jump to instruction i if the contents of the registers R_m and R_n are the same. Otherwise execute the next instruction

We have used the syntax of the imperative fragment of SML to describe the effect of each instruction on the appropriate register. The semantics of the jump instruction mixes pidgin SML with the notation for conditional expressions used in languages like C and Perl. The verbal description describes the semantics informally in English.

- Given a URM program $P = I_1, I_2, \ldots, I_p$, register contents $r_1, r_2, \ldots$ respectively and a program counter $PC \in \mathbb{P}$ denoting the next instruction to be executed, a **configuration** of the URM is a 3-tuple $\langle P, M, PC\rangle$ where $M(n) = r_n$.

- Given an initial memory M_0 and a program $P = I_1, I_2, \cdots, I_p$, with $p \geq 1$, the initial configuration of the URM is

$$\gamma_0 = \langle P, M_0, 1\rangle$$

- Let $\gamma = \langle P, M, j\rangle$ be any configuration of the URM. The URM is said to **halt** if $j > p$. Such a configuration is also called the **halting configuration** and the result of the computation is $M(1)$ the contents of the register R_1. If $0 < j \leq p$, then a **step** of the computation from γ is given by $\gamma \mapsto \gamma'$ where $\gamma' = \langle P, M', j'\rangle$ and M' and j' are as given by the following table:

$I_j =$	$M' =$	$j' =$	
$\mathsf{Z}(n)$	$\{0/n\}M$	$j+1$	
$\mathsf{S}(n)$	$\{M(n)+1/n\}M$	$j+1$	
$\mathsf{C}(m,n)$	$\{M(m)/n\}$	$j+1$	
$\mathsf{J}(m,n,i)$	M	i	if $M(m) = M(n)$
	M	$j+1$	if $M(m) \neq M(n)$

where $M' = \{a/n\}M$ denotes that M' is identical to M for all values other than n and $M'(n) = a$ regardless of whatever $M(n)$ might be.

- **Unconditional jumps** are described by instructions such as $\mathsf{J}(n,n,j)$ since $M(n) = M(n)$ always. Similarly **skip** instructions (or **no-op** as they are referred to in hardware) may be implemented using instructions such as $\mathsf{C}(n,n)$ or by jumping unconditionally to the next instruction.

- The URM executes instructions in sequence (unless it executes a jump instruction, when the sequence is altered) till the instruction referred to by PC does not exist. Then the URM is said to halt.

Definition 2.1 *A memory map M of the URM is said to be* **finite** *if there are only a finite number of registers containing non-zero values i.e. the set $\{M(n) \mid M(n) > 0\}$ is finite.*

If the (initial) memory M_0 in the initial configuration is such that $\{M_0(n) \mid M_0(n) > 0\}$ is finite, then the result of executing each instruction results in a new memory state M' such that $\{M'(n) \mid M'(n) > 0\}$ is also finite. In other words, at no stage of the execution of the program does the URM require more than a finite number of registers to be used.

Lemma 2.1

1. *Each URM instruction preserves finite memory maps.*
2. *The only registers whose contents may modified by a URM program are those that are explicitly named in the program.*

Every instruction has a **deterministic** effect on the URM machine. In other words, given a memory map M each of the instructions $\mathsf{Z}(n)$, $\mathsf{S}(n)$ and $\mathsf{C}(m, n)$ when executed, would yield a *unique* memory state M' in all programs and under all execution conditions. The instruction $\mathsf{J}(m, n, i)$ too has the effect of *uniquely* determining the next instruction (if any) to be executed.

Lemma 2.2 .

1. *Every non-halting configuration of the URM has a unique successor. That is, $\gamma \mapsto \gamma'$ and $\gamma \mapsto \gamma''$ implies $\gamma' = \gamma''$.*
2. *If Γ is the set of all URM configurations with finite memory maps then $\mapsto$ is a partial function from Γ to Γ.*

$\mapsto$ is a partial function on the configurations of the URM because it is undefined for halting configurations. It suffices for us to consider only configurations with finite memory maps, since we are interested in the notion of computability which requires for each task only a finite amount of resources such as storage and program length. However, we do need to consider the availability of unlimited storage. If we did fix storage limits to some fixed number then our theoretical development suffers when technology improves to provide more storage than our limit. A theory such as ours would have long term applicability only if it does not artificially constrain the solutions to problems. Similarly while we consider the executions of URM programs of arbitrary length our fniteness constraints are limited to specifying that the program may not be of infinite length.

The reader could ask whether the limitation to just four kinds of instructions is not an artificial constraint. Our answer to that would be two-fold viz.

1. that only the *kinds* of operations are being fixed and that any reasonable mechanical device (going by the history of mechanical devices) would be capable of performing only a fixed finite set of different *kinds* of operations, and
2. the actual parameters that these operations may take is actually infinite.

Definition 2.2 *Let γ_0 be an initial configuration of the URM. A (finite or infinite) sequence of the form $\gamma_0 \mapsto \gamma_1 \mapsto \cdots$ is called an* **execution sequence** *from γ_0. A* **computation** *of the URM is a maximal execution sequence from an initial configuration.*

Lemma 2.3 *Let $\gamma_0 \mapsto \gamma_1 \mapsto \gamma_2 \mapsto \cdots$ be an execution of the URM with the corresponding memory maps $M_0, M_1, M_2, \ldots$. If M_0 is a finite memory map, then so are each of the succeeding memory maps M_1, M_2, etc.*

Definition 2.3

1. *A URM program P* **converges** *on input $(a_1, a_2, ..., a_k)$ if it begins execution from the initial memory state $M_0 = a_1, a_2, ..., a_k, 0, 0, ...$ and $PC = 1$ and halts after a finite number of steps of computation.*
   Notation. $P(a_1, a_2, ..., a_k) \downarrow$ denotes that P converges on input $(a_1, a_2, ..., a_k)$
2. *Otherwise P is said to* **diverge** *on the input. Notation. $P(a_1, a_2, ..., a_k) \uparrow$.*
3. *P is said to* **converge to** *$b \in \mathbb{N}$ on the input $(a_1, a_2, ..., a_k)$ if it converges on the input and reaches a configuration in which $M(1) = b$.*
   Notation. $P(a_1, a_2, ..., a_k) \downarrow b$ denotes that P converges on input $(a_1, a_2, ..., a_k)$ to b.

Definition 2.4 *A URM program P* **URM-computes** *a partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, if and only if*

- *for each $(a_1, ..., a_k) \in \mathsf{Dom}(f)$, $P(a_1, ..., a_k) \downarrow f(a_1, ..., a_k)$ and*
- *$P(a_1, ..., a_k) \uparrow$ for all $(a_1, ..., a_k) \notin \mathsf{Dom}(f)$*

Definition 2.5 *A partial function f is* **URM-computable** *if and only if there is a URM program P which computes f. A predicate p is* **URM-decidable** *if and only if there is a URM program which computes its characteristic function χ_p.*

For each $k \geq 0$, every URM program P computes a partial function $\phi_P^{(k)} : \mathbb{N}^k \rightharpoonup \mathbb{N}$. Given an input $(a_1, a_2, ..., a_k)$ it begins execution from the initial memory state $M_0 = a_1, a_2, \ldots, a_k, 0, 0, \ldots$ and $PC = 1$ and if and when it halts $\phi_P^{(k)}(a_1, ..., a_n) = !R_1$, i.e. the final result is the value in register R_1. If it does not halt for the given input then $\phi_P^{(k)}(a_1, ..., a_n) = \perp$. In the case decidability however, if p is an m-ary relation for some $m > 0$, note that the characteristic function is always total and hence a URM program that implements it should halt for every m-tuple input. But it is quite possible that the program does not halt on some $(m+1)$-tuple input.

2.2 Examples

In the following examples we illustrate programming the URM. Since the actual programs can be quite tedious and confusing to read and understand, we also provide flow-charts to explain the working of the programs. The language of flow charts consists of boxes, circles, diamonds and arrows connecting them to form a flow-chart program. We follow the usual convention that boxes denote actions that could change the memory state.

Boxes enclose actions which may change the state of the memory. We use a notation drawn from pidgin SML for this purpose. The same notation has been used earlier in the tables defining the semantics of the instruction set. Each box may enclose a sequence of instructions.

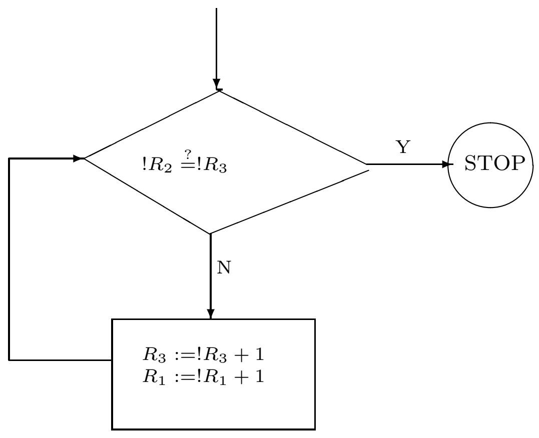
Diamonds indicate decisions to be taken depending on the evaluation of the condition in the diamond. Each diamond encloses a condition of the form $!R_m \overset{?}{=} !R_n$ whose truth in the memory state requires a different flow of control. The letters "Y" and "N" on the arrows emanating from a diamond indicate the flow of control when the condition is true or false respectively.

Circles usually enclose the single word "STOP" and indicate that a halting configuration has been reached and no further execution is possible or necessary.

In each case the initial memory map is also indicated

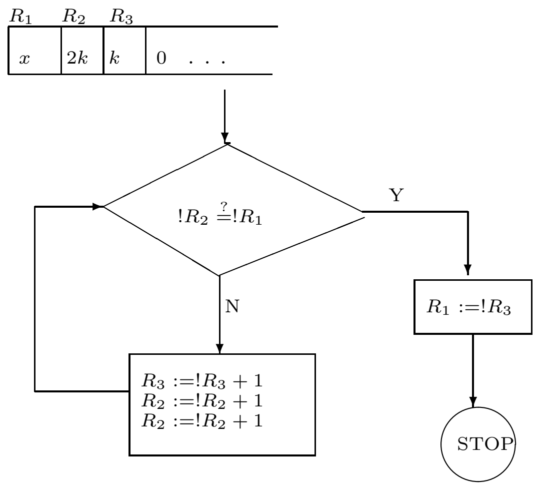
Example 2.2.1 *The addition function $x + y$.*

R_1	R_2	R_3	
x	y	0	. . . .



```
1. J ( 2, 3, 5 )
2. S ( 3 )
3. S ( 1 )
4. J ( 1, 1, 1)
```
P_1

Example 2.2.2 $f(x) = \begin{cases} \frac{1}{2}x & \textit{if } x \textit{ is even} \\ \perp & \textit{otherwise} \end{cases}$



```
1. J( 2,1,6 )
2. S ( 3 )
3. S ( 2 )
4. S( 2 )
5. J ( 1, 1, 1)
6. C ( 3, 1 )
```
P_2

Further notes, notations and terminology

- Every instruction of the URM is deterministic (in the sense that the effect of the instruction on the state of the program is unique). Therefore every URM-program is computes partial of the initial state to the final state

- Since we are interested only in the contents of $R1$ when a program P halts, every program is a partial function whose range is contained in $\mathbb{N}$.

  <u>Notation.</u> Given a program P and a tuple of inputs $\overrightarrow{a} = (a_1, \ldots, a_k)$, $P(a_1, \ldots, a_k)$ denotes the result of executing P with the input $\overrightarrow{a}$

- The type of the function determined by a program P depends entirely on the vector of inputs that we consider.

  1. The addition program P_1 in the example does not necessarily represent only the binary operation ofaddition.
  2. If there are no inputs it implements the constant function 0.
  3. If the initial configuration has only a single input, x then P_1 implements the identity function $f(x) = x$.
  4. If the initial cofiguration consists of 3 values then P_1 implements the partial function

$$f(x, y, z) = \begin{cases} x + y - z & \text{if x} \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

Hence for any URM program P and for any $k \geq 0$, P implments a unique partial function $\phi_P^k : \mathbb{N}^k \not\longrightarrow \mathbb{N}$ i.e the arity of the function implemented by a program is not pre-determined and is dependent entirely on the length of the input vector.

Definition 2.6 *A URM-program P implements a function $g : \mathbb{N}^k \not\longrightarrow \mathbb{N}$ iff $\phi_P^k = g$.*

Example 2.2.3 *Consider program P_2. The following are the functions it implements for for the arities 1,2 and 3 respectively.*

$$\phi^{(1)}_{P_2}(x) = \begin{cases} \frac{1}{2}x & \textit{if } x \textit{ is even} \\ \perp & \textit{otherwise} \end{cases}$$

$$\phi^{(2)}_{P_2}(x,y) = \begin{cases} \frac{1}{2}(x-y) & \textit{if } x \textit{ is even(x-y)} \\ \perp & \textit{otherwise} \end{cases}$$

$$\phi^{(3)}_{P_2}(x,y,z) = \begin{cases} z + \frac{1}{2}(x-y) & \textit{if } x \textit{ is even} \\ \perp & \textit{otherwise} \end{cases}$$

Notes.

1. Each program P implements an <u>infinite</u> number of <u>distinct</u> functions and <u>exactly one</u> function of <u>each</u> arity.
2. For each function $f : \mathbb{N}^k \longrightarrow\!\!\!\!\!\!+\;\; \mathbb{N}$ that has an implementation P, there are an <u>infinite</u> number of <u>distinct</u> programs that implement the same function.

Example 2.2.4 *Add to program P_2 the following instruction*

$J(1,1,8)$

This produces a new program $P_2{}'$, which implements exactly the same function as P. We could similarly add more instructions and keep getting syntactically distinct programs which all implement the same function.

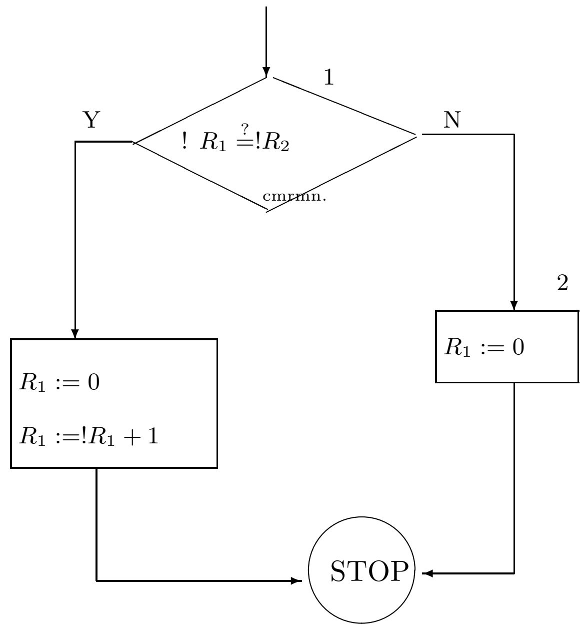
URM -Decidability

Definition 2.7 *A predicate (or a relation) $p \subseteq \mathbb{N}^m$ is URM-decidable if its characteristic (total) function $\chi_p : \mathbb{N}^m \rightarrow \{0,1\}$ is URM-computable.*

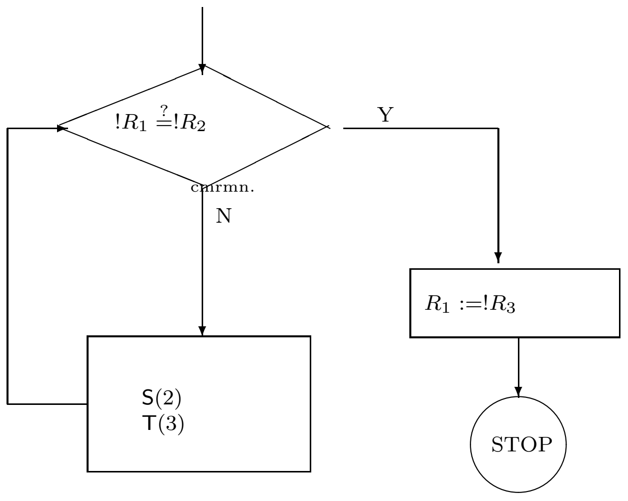
Example 2.2.5 *The following predicates are URM-decidable.*

1. *$x = y$ is implemented by*

```
1. J ( 1, 2, 4)
2. Z ( 1 )
3. J ( 1, 1, 6 )
4. Z ( 1 )
5. S ( 1 )
```



2. odd(x) is implemented by incrementing R_2 and toggling R_3 to reflect the evenness or oddness of R_2. Let $T(3)$ denote toggle R_3. Initially $!R_2 = 0$ and $!R_3 = 0$. The final flow chart including $T(3)$ looks as follows.



cmrmn.

The final program is then as follows.

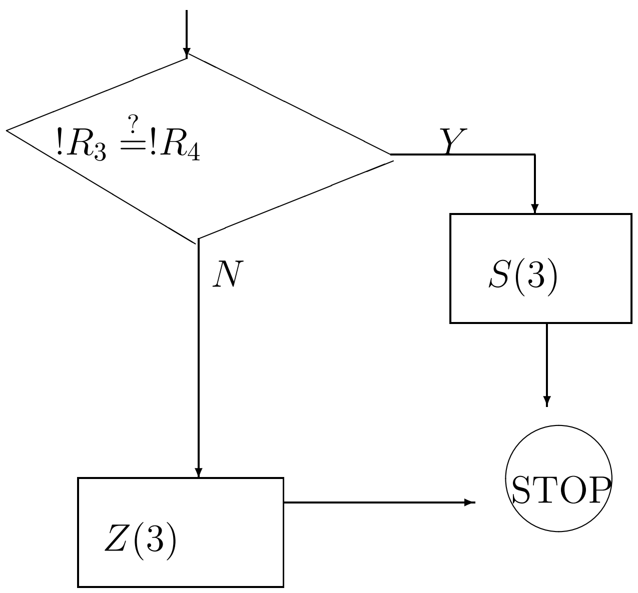


Figure 2.1: Implementing $\mathsf{T}(3)$

```
1. J (1,2,8)
2. S (2)
3. J (3,4,6)
4. Z (3)
5. J (1,1,1)
6. S (3)
7. J (1,1,1)
8. C (3,1)
```

Composing URM-Programs

Function composition in URM is simply the catenation of the component programs. However we cannot catenate two programs to produce a composition unless the termination or halting of the first is tackled in a meaningful fashion – clearly then jump instructions have to be standardized.

Definition 2.8 *A program $P = I_1, ..., I_r$ is in* **standard form** *provided every jump instruction $J(m, n, p)$ is such that $1 \leq p \leq r + 1$.*

Lemma 2.4 *For any program P there is a program $P^\star$ in standard form such that for all input vectors $\overrightarrow{a}$, $P(\overrightarrow{a}) \downarrow b$ iff $P^*(\overrightarrow{a}) \downarrow b$.*

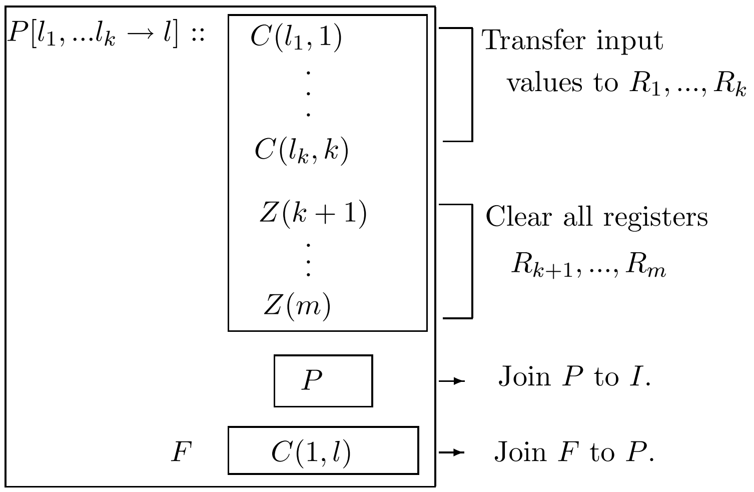
Proof: If P consists of p instructions then P^* is the program obtained from P by replacing all jump instructions that go outside the program to jump to $p+1$. □

Definition 2.9 *For any two programs P, Q in standard form with $P = I_1, \ldots, I_p$ and $Q = I'_1, \ldots, I'_q$, the* **join** *of P and Q is the program $PQ = I_1, ..., I_p, I_{p+1}, ..., I_{p+q}$ where any jump instruction $I'_j = J(m, n, r)$ is replaced by $I_{p+j} = J(m, n, r+p)$ and for all other instructions $I'_j = I_{p+j}$.*

Macros: The operation $[l_1, \ldots, l_k \rightarrow l]$

Our convention of reserving the first few registers for input and the first register for output has the disadvantage that it is inconvenient when joining several smaller programs together to obtain a larger program. While composing programs P and Q together it often happens that the inputs to some later program in the join may lie scattered in different registers. Similarly the output of some component program may have to be stored in some register other than R_1 in order that the contents of these registers do not affect or interfere with the computations of the main program.

Let P be a URM program in standard form and k such that $\phi_P^{(k)} : \mathbb{N}^k \!+\!\!\longrightarrow \mathbb{N}$ is computed by P. Since P is a finite sequence of instructions there exists a highest index of register that P uses. Let $\rho(P)$ denote the index of this register. Further let $l_1, l_2, ..., l_n$, l be distinct registers with each index exceeding $\rho(P)$. Then the program $P' = P[l_1, ..., l_k \rightarrow l]$ denotes a new program whose inputs are in registers $R_{l_1}, ..., R_{l_k}$ and output (if any) in register R_l. Let $m = max(\rho(P), l_1, \ldots, l_k, l)$. This new program P' is depicted in the figure below


$P[l_1, ...l_k \rightarrow l]$ ::
$C(l_1, 1)$
$C(l_k, k)$
$Z(k+1)$
$Z(m)$
P
F
$C(1, l)$
Transfer input values to $R_1, ..., R_k$
Clear all registers $R_{k+1}, ..., R_m$
Join P to I.
Join F to P.

URM-Computability on other Domains

Definition 2.10 *A* **coding** *of a domain D is an explicit and effective injection $\alpha : D \rightarrow \mathbb{N}$. A function $f : D \nrightarrow D$ is said to be URM-computable under the coding α if there exists a URM-computable function $f^* : \mathbb{N} \nrightarrow \mathbb{N}$ such that the following diagram holds.* i.e

$$\begin{tikzcd} D \arrow[r, "f", "+"'] \arrow[d, "\alpha"'] & D \\ \mathbb{N} \arrow[r, "f^\star"', "+"] & \mathbb{N} \arrow[u, "\alpha^{-1}"'] \end{tikzcd}$$

For any $d \in D$, $f(d) = \alpha^{-1}(f^\star(\alpha(d)))$ In other words $\boxed{f = \alpha \circ f^\star \circ \alpha^{-1}}$

Note:

1. α needs to be injective and total.
2. α may not be surjective, so α^{-1} is partial in general.

Chapter 3

The Primitive Recursive Functions

3.1 Introduction

Partial recursive functions are a different approach to computability. In this and the next chapter we will see the relation between partial recursive functions and URM-computability. However we need to first define the class of primitive recursive functions and then gradually build up to the partial recursive functions.

Definition 3.1 *The* **initial** *functions are*

1. *the* constant 0*,*
2. *the unary* successor *function s*
3. *the family $\{\pi_i^n \mid n \in \mathbb{P}, 1 \leq i \leq n\}$ of* projection *functions, where for each n-tuple $(x_1, \ldots, x_n)$, and each i, $1 \leq i \leq n$, $\pi_i^n(x_1, \ldots, x_n) = x_i$*

URM and μ-recursive functions

Lemma 3.1 *The initial functions are all URM-computable.*

Proof:

- *the constant 0 is computed by the program* $\mathsf{Z}(1)$
- *the unary successor function s is computed by the program* $\mathsf{S}(1)$
- *For each $n \in \mathbb{P}$ and i, $1 \leq i \leq n$, π_i^n is computed by the program* $\mathsf{C}(i, 1)$*.*

□

Lemma 3.2 Composition is URM-computable. *Let $f : \mathbb{N}^k \longmapsto \mathbb{N}$, $g_1, \cdots, g_k : \mathbb{N}^m \longmapsto \mathbb{N}$ Then the composition of f and $\overrightarrow{g}$ denoted by $f \circ \overrightarrow{g}$ is also URM-computable.*

Proof: Let f and $g_1, \cdots, g_k$ be computed by programs $F, G_1, ..., G_k$ respectively and let $n = max(m, k, \rho(F), \rho(G_1), ..., \rho(G_k))$. The following program computes $f \circ \overrightarrow{g}$

```
C(1, m+1)
⋮
C(n, m+n)
G₁[m+1, m+2, ..., m+n → m+n+1]
⋮
Gₖ[m+1, m+2, ..., m+n → m+n+k]
F[m+n+1, ..., m+n+k → 1]
```

□

Often we may require to obtain new functions from existing functions by permuting the arguments, or by identifying some arguments or by the addition of dummy arguments. Typical examples of such functions are as follows:

Permuting arguments Consider the "divisor of" relation on numbers expressed in terms of its characteristic function $divisorof : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$. We may define the "multiple of" relation $multipleof : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ in terms of the $divisorof$ function by permuting the two arguments, i.e. $multipleof(x, y) = divisorof(y, x)$.

Identification of arguments Consider the function $double : \mathbb{N} \to \mathbb{N}$ which is obtained from addition function by identifying the two operands of addition.

Addition of dummy arguments Any constant $k \in \mathbb{N}$ may be elevated to the status of a function such as $k^{(1)} : \mathbb{N} \to \mathbb{N}$ defined as $k^{(1)}(x) = k$ and further to $k^{(2)} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined as $k^{(2)}(x, y) = k$.

The following theorem shows that these operations are also URM-computable and expressed using only composition and the initial functions.

Theorem 3.3 *Let $f : \mathbb{N}^k \longmapsto \mathbb{N}$ be a URM-computable function defined in terms of the variables $x_1, \ldots, x_k$. Let $x_{i_1}, \ldots, x_{i_n}$ be a sequence of n-variables drawn from the set $\{x_1, x_2, ..., x_m\} \supseteq \{x_1, \ldots x_k\}$. Then the function $h : \mathbb{N}^n \longmapsto \mathbb{N}$ defined by*

$$h(x_{i_1}, \ldots, x_{i_n}) = f(x_{j_1}, \ldots, x_{j_k})$$

where $\{x_{j_1}, \ldots, x_{j_k}\} \subseteq \{x_1, x_2, ..., x_m\}$*, is also URM- computable.*

Proof: Writing $\overrightarrow{x}^n$ for $(x_{i_1}, ..., x_{i_n})$ and $\overrightarrow{\pi^m}^k$ for $(\pi^m_{j_1}, \cdots, \pi^m_{j_k})$ and using composition we have

$$h(\overrightarrow{x}^n) = [f \circ \overrightarrow{\pi^m}^k](x_1, \ldots, x_m)$$

Since composition and projection are both URM-computable it follows that h is also URM-computable. □

Notes.

1. The case of the empty sequence () is taken care of by lemma 3.7 which shows that the naturals are all computable.
2. Since $x_{i_1}, \ldots, x_{i_n}$ are all drawn from the set $\{x_1, x_2, ..., x_m\} \supseteq \{x_1, \ldots x_k\}$, some of the arguments of h may be identifications of variables in $\{x_1, \ldots x_k\}$, some may be reorderings (permutations) of the variables in $\{x_1, \ldots x_k\}$ and some of them may be drawn from $\{x_1, x_2, ..., x_m\} - \{x_1, \ldots x_k\}$ (these are the dummy arguments of h). Notice that the arguments $(x_{j_1}, \ldots, x_{j_k})$ f in the definition of h are entirely drwan from $\{x_1, x_2, ..., x_m\}$.

3.2 Primitive Recursion

Given total functions $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$, consider the following relation $h \subseteq \mathbb{N}^{n+2}$ where $h = \bigcup_{i \geq 0} h_i$ and the individual h_i , $i \geq 0$ are defined by induction as follows.

$$\begin{aligned} h_0 &= \{(\overrightarrow{x}, 0, f(\overrightarrow{x})) \mid \overrightarrow{x} \in \mathbb{N}^n\} \\ h_{i+1} &= \{(\overrightarrow{x}, i+1, z_{i+1}) \mid \overrightarrow{x} \in \mathbb{N}^n, z_{i+1} = g(\overrightarrow{x}, y, z_i), (\overrightarrow{x}, y, z_i) \in h_i\} \quad \text{for } i \geq 0 \end{aligned}$$

In fact as the following theorem shows we may express the relation h by a pair of so called **recursion equations** which uniquely define h in terms of the functions f and g.

$$h(\overrightarrow{x}, 0) = f(\overrightarrow{x}) \tag{3.1}$$

$$h(\overrightarrow{x}, y+1) = g(\overrightarrow{x}, y, h(\overrightarrow{x}, y)) \tag{3.2}$$

Theorem 3.4 *Given total functions $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$, there exists a unique total function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying the recursion equations (3.1) and (3.2).*

Proof:

Existence. Consider sets of $(n+2)$-tuples of numbers. A set $S \subseteq \mathbb{N}^{n+2}$ is called **satisfactory** with respect to the recursion equations (3.1) and (3.2) if for every choice of $\overrightarrow{a} \in \mathbb{N}^n$,

Basis. $(\overrightarrow{a}, 0, f(\overrightarrow{a})) \in S$ and

Induction. if $(\overrightarrow{a}, b, c) \in S$, then $(\overrightarrow{a}, b+1, g(\overrightarrow{a}, b, c)) \in S$.

Let $\mathcal{S}$ be the set of all satisfactory sets. Clearly $\mathcal{S} \neq \emptyset$ since $\mathbb{N}^{n+2}$ is satisfactory. Consider $S_0 = \bigcap \mathcal{S}$. S_0 is also non-empty since it contains at least every tuple of the form $(\overrightarrow{a}, 0, f(\overrightarrow{a}))$. Further

Claim 1. *S_0 is satisfactory and is contained in every satisfactory set.*

Claim 2.

For each choice of $\overrightarrow{a}$ and b, there is a unique c such that $(\overrightarrow{a}, b, c) \in S_0$.

$\vdash$ Suppose there is a tuple $(\overrightarrow{a}, 0, c) \in S_0$. If $c \neq f(\overrightarrow{a})$ then $S_0' = S_0 - \{(\overrightarrow{a}, 0, c)\}$ would be satisfactory since $(\overrightarrow{a}, 0, f(\overrightarrow{a})) \in S_0'$ but $S_0 \not\subseteq S_0'$. Hence $c = f(\overrightarrow{a})$. Assume the claim is true for all $0 \leq b' \leq b$. Then for each choice of $\overrightarrow{a}$ there exists a unique $c \in \mathbb{N}$ such that $(\overrightarrow{a}, b, c) \in S$. Since S_0 is satisfactory it follows that $(\overrightarrow{a}, b+1, g(\overrightarrow{a}, b, c)) \in S_0$. Suppose there exists $(\overrightarrow{a}, b+1, d) \in S_0$ such that $d \neq g(\overrightarrow{a}, b, c)$. Then again $S_0' = S_0 - \{(\overrightarrow{a}, b+1, d)\}$ is a satisfactory set and $S_0 \not\subseteq S_0'$ which is impossible. $\dashv$

Uniqueness. Suppose there exist two functions h and h' both of which satisfy the recursion equations (3.1) and (3.2). Then from the fact that f and g are (total) functions we have

$$\forall \overrightarrow{a}\ [h(\overrightarrow{x}, 0) = f(\overrightarrow{x}) = h'(\overrightarrow{x}, 0)]$$

and assuming that for all $\overrightarrow{a}$ and some $b \geq 0$,

$$h(\overrightarrow{a}, b) = h'(\overrightarrow{a}, b)$$

we have

$$\begin{aligned} h(\overrightarrow{a}, b+1) &= g(\overrightarrow{a}, b, h(\overrightarrow{a}, b)) \\ &= g(\overrightarrow{a}, b, h'(\overrightarrow{a}, b)) \\ &= h'(\overrightarrow{a}, b+1) \end{aligned}$$

□

We may extend the above result to partial functions by insisting that functions be strict with respect to the undefined.

Theorem 3.5 *Given partial functions $f : \mathbb{N}^n \not\longrightarrow \mathbb{N}$ and $g : \mathbb{N}^{n+2} \not\longrightarrow \mathbb{N}$, there exists a unique partial function $h : \mathbb{N}^{n+1} \not\longrightarrow \mathbb{N}$ satisfying the recursion equations (3.1) and (3.2).*

Proof: We leave the details of the proof of the above theorem to the reader, while simply noting that if f is the everywhere undefined function then so is h (prove it!). Further in the inductive definitions of the relations h_i, if for any $i \geq 0$, $h_i = \emptyset$ then $h_j = \emptyset$ for all $j > i$. □

We may regard both *function composition* and *primitive recursion* as (higher order) operators which take functions of appropriate arities as arguments and yield new functions as results. Both composition and primitive recursion are *polymorphic* functions. More precisely, our notation for functions f of type $\mathbb{N}^k \not\longrightarrow \mathbb{N}$ simply specifies that f is simply a member of the set $\mathbb{N}^k \not\longrightarrow \mathbb{N}$, which is exactly the space of all partial functions of the specified type. That is f is simply a point in the space $\mathbb{N}^k \not\longrightarrow \mathbb{N}$ of functions of arity k which yield natural numbers as values. Function composition and primitive recursion are therfore merely functions or operators on such spaces.

In the particular case when a function f of arity $k > 0$ is composed with k functions $\overrightarrow{g}$, where each of the components of $\overrightarrow{g}$ is of arity $m > 0$, we get a new function h of arity m. In such a case, the type of composition may be described as follows.

$$\circ : (\mathbb{N}^k \not\longrightarrow \mathbb{N}) \times (\mathbb{N}^m \not\longrightarrow \mathbb{N})^k \longrightarrow (\mathbb{N}^m \not\longrightarrow \mathbb{N})$$

Similarly primitive recursion may be regarded as a polymorphic operator of type

$$\mathbf{pr} : (\mathbb{N}^n \not\longrightarrow \mathbb{N}) \times (\mathbb{N}^{n+2} \not\longrightarrow \mathbb{N}) \not\longrightarrow (\mathbb{N}^{n+1} \not\longrightarrow \mathbb{N})$$

Notation

- The *composition* of f with $\overrightarrow{g}$ is denoted $[\overrightarrow{g}\,; f]$ or $[f \circ \overrightarrow{g}]$.
- The function defined by primitive recursion on f and g is denoted $[f \ \mathbf{pr}\ g]$.

Primitive recursion is URM-computable

Definition 3.2 *A function $h : \mathbb{N}^{n+1} \not\longrightarrow \mathbb{N}$ is said to be defined by primitive-recursion on functions $f : \mathbb{N}^n \not\longrightarrow \mathbb{N}$ and $g : \mathbb{N}^{n+2} \not\longrightarrow \mathbb{N}$ if it satisfies the equations*

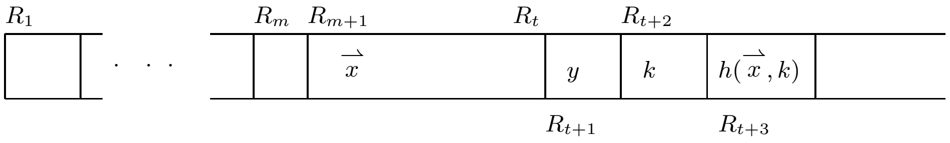
$$\begin{array}{rcl} h(\overrightarrow{x}, 0) & = & f(\overrightarrow{x}) \\ h(\overrightarrow{x}, y+1) & = & g(\overrightarrow{x}, y, h(\overrightarrow{x}, y)) \end{array}$$

Theorem 3.6 *If $f : \mathbb{N}^n \not\longrightarrow \mathbb{N}$ and $g : \mathbb{N}^{n+2} \not\longrightarrow \mathbb{N}$ are URM-computable then so is $h : \mathbb{N}^{n+1} \not\longrightarrow \mathbb{N}$ with*

$$\begin{array}{rcl} h(\overrightarrow{x}, 0) & = & f(\overrightarrow{x}) \\ h(\overrightarrow{x}, y+1) & = & g(\overrightarrow{x}, y, h(\overrightarrow{x}, y)) \end{array}$$

Proof: Let F and G compute f and g resp and let $m = max(n+2, \rho(F), \rho(G))$ and $t = m+n$

Then the following program computes $h(\overrightarrow{x}, y)$.



```
    C(1, m+n)
    C(n+1, t+1)
    F[1, 2, ..., n → t+3]
p.  J(t+2, t+1, q)
    G[m+1, ..., m+n, t+2, t+3 → t+3]
    S(t+2)
    J(1, 1, p)
q.  C(t+3, 1).
```

□

Lemma 3.7 *Every natural number is URM-computable*

Proof: Each natural number $n : \emptyset \longrightarrow \mathbb{N}$ is a total function from the empty set to the naturals.

$\boxed{\mathsf{J}(1,1,2)}$ computes the number 0.

For any $n > 0$ the following program computes it

```
1.   S(1)
.    .
.    .
.    .
n.   S(1)
```

□

Definition 3.3 *The class PRIMREC of* **primitive recursive** *functions is the smallest class of functions containing the* initial functions *and which is closed under the operations of* function composition *and* primitive recursion

That is, the class PRIMREC may be defined by induction as follows

$$\begin{aligned} PRIMREC_0 &= \{0\} \cup \{s\} \cup \{\pi_i^n \mid n > 0, 1 \leq i \leq n\} \\ PRIMREC_{j+1} &= PRIMREC_j \cup \{[f \circ \overrightarrow{g}] \mid f, \overrightarrow{g} \in PRIMREC_j\} \\ &\quad \cup \{[f \ \mathbf{pr} \ g] \mid f, g \in PRIMREC_j\} \\ PRIMREC &= \bigcup_{k \geq 0} PRIMREC_k \end{aligned}$$

In the above definition, we have not explicitly mentioned the type restrictions on composition and primitive recursion, for the sake of conciseness. However, these restrictions are required.

Hence in order to show that a certain function belongs to the class $PRIMREC$, we require to show the construction of the function as belonging to a certain $PRIMREC_k$, for some $k \geq 0$. This implies that we need to show an explicit construction of the function using only the initial functions and the operations of composition and primitive recursion, while still respecting the type restrictions on composition and primitive recursion.

Lemma 3.8 *Every function in $PRIMREC$ is total.*

Proof outline: By induction on the induction variable j in definition 3.3. It follows easily since the initial functions are total and each application of composition or primitive recursion is on total functions belonging to a class with a smaller index. $\square$

Lemma 3.9 *Every function defined by composition or by primitive recursion on functions in $PRIMREC$ is also primitive recursive.*

Notice that the proof of theorem 3.3 may be reproduced as the proof of lemma 3.10.

Lemma 3.10 *Let $f : \mathbb{N}^k \to \mathbb{N}$ be a primitive recursive function defined in terms of the variables $x_1, \ldots, x_k$. Let $x_{i_1}, \ldots, x_{i_n}$ be a sequence of n-variables drawn from the set $\{x_1, x_2, ..., x_m\} \supseteq \{x_1, \ldots x_k\}$. Then the function $h : \mathbb{N}^n \to \mathbb{N}$ defined by*

$$h(x_{i_1}, \ldots, x_{i_n}) = f(x_{j_1}, \ldots, x_{j_k})$$

where $\{x_{j_1}, \ldots, x_{j_k}\} \subseteq \{x_1, x_2, ..., x_m\}$, is also primitive recursive.

Theorem 3.11 *The following <u>underlined</u> functions are primitive recursive (and hence URM-computable).*

Proof outline: In each case we express the function as being obtained by a finite number of applications of composition and primitive recursion on the initial functions. Often we simply express a function in terms of composition or primitive recursion on functions shown earlier to be in the class $PRIMREC$. These formulations not only prove that they are primitive recursive, they show that these functions are URM-computable as well (since we have shown that both composition and primitive recursion are URM computable operations. In each case, the proof is written between the brackets $\vdash \cdots \dashv$.

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

   $\vdash$ 0 is initial. For $k > 0$ we have $\boxed{k = [\underbrace{s \circ \cdots \circ s}_{k\text{-times}}](0)}$.

   $\dashv$

2. $id(x) = x$.

   $\vdash$ Clearly $\boxed{id = \pi_1^1}$. $\dashv$

   .

3. $add(x, y) = x + y$.

   $\vdash$ Using the induction on y we get $x + 0 = x = id(x)$ and $x + (y + 1) = (x + y) + 1 = s(x + y) = s(add(x, y)) = [s \circ add]$. We then have $\boxed{add = [id\ \mathbf{pr}\ [s \circ \pi_3^3]]}$. $\dashv$

4. $mult(x, y) = xy$.

   $\vdash$ Again by induction on y we have $x.0 = 0$ and $x.(y + 1) = xy + x = x + xy = x + mult(x, y) = add(x, mult(x, y))$. By lemma 3.10 we may express the constant 0 also as a constant function (of any number of arguments). We write it as $\mathbf{0}(x)$ and it is clearly primitive recursive. The function $g(x, y, z) = add(x, z) = add \circ (\pi_1^3, \pi_3^3)$ then yields the definition $\boxed{mult = [\mathbf{0}\ \mathbf{pr}\ [add \circ (\pi_1^3, \pi_3^3)]]}$.

   $\dashv$

5. $texp(x, y) = x^y$ (assuming $0^0 = 1$).

   $\vdash$ Using the definitions $texp(x, 0) = 1 = s(0) = s(\mathbf{0}(x))$ and $texp(x, y+1) = mult(x, texp(x, y))$ we use a reasoning similar to that for multiplication to get $\boxed{texp = [[s \circ \mathbf{0}]\ \mathbf{pr}\ [mult \circ (\pi_1^3, \pi_3^3)]]}$.
   $\dashv$

6. $tpred(x) = x - 1$ (assuming $tpred(0, 1) = 0$).

   $\vdash$ Here x is the induction variable. So we get $\boxed{tpred = [0\ \mathbf{pr}\ \pi_1^2]}$ $\dashv$

7. $monus(x, y) = x \dot{-} y$.

   $\vdash$ We have $x \dot{-} 0 = x = id(x)$ and $x \dot{-} (y+1) = tpred(x \dot{-} y)$ Hence $\boxed{monus = [id\ \mathbf{pr}\ [tpred \circ \pi_3^3]]}$
   $\dashv$

8. $sg(x)$ defined as $sg(x) = \begin{cases} 0 & \text{if x=0} \\ 1 & \text{if } x > 0 \end{cases}$

$\vdash$ Clearly $sg(0) = 0$ and $sg(x+1) = 1$. It is easy to see that $sg(x) = 1 \dot{-} (1 \dot{-} x) = monus(1, monus(1, x))$. Using 1 as unary constant function we have

$\boxed{sg = [monus \circ (1, [monus \circ (1, id)])]}$ $\dashv$

9. $\overline{sg}(x)$ defined as $\overline{sg}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$

   $\vdash$ $\boxed{\overline{sg}(x) = 1 \dot{-} sg(x)}$ $\dashv$

10. $sdiff(x, y) = |x - y|$.

    $\vdash$ $|x - y| = (x \dot{-} y) + (y \dot{-} x) = add(monus(x, y), monus(y, x))$. Hence

    $\boxed{sdiff = [add \circ ([monus \circ (\pi_1^2, \pi_2^2)], [monus \circ (\pi_2^2, \pi_1^2)])]}$. $\dashv$

11. $fact(x) = x!$.

    $\vdash$ We have $0! = 1$ and $(x+1)! = x! \times (x+1) = mult(s(x), fact(x))$, which yields $\boxed{fact = [1 \ \mathbf{pr} \ [mult \circ ([s \circ \pi_1^2], \pi_2^2)]]}$ $\dashv$

12. $min(x, y)$.

    $\vdash$ $min(x, y) = x \dot{-} (x \dot{-} y)$ Hence $\boxed{min = [monus \circ (\pi_1^2, [monus \circ (\pi_1^2, \pi_2^2)])]}$ $\dashv$

13. $max(x, y)$.

    $\vdash$ $max(x, y) = y + (x \dot{-} y)$. Hence $\boxed{max = [add \circ (\pi_1^2, [monus \circ (\pi_1^2, \pi_2^2)])]}$ $\dashv$

14. $trem(x, y) = \begin{cases} y & \text{if } x = 0 \\ y \ \mathbf{mod} \ x & \text{otherwise} \end{cases}$

    $\vdash$ We have

$$trem(x, y+1) = \begin{cases} 0 & \text{if } trem(x, y) + 1 = x \\ trem(x, y) + 1 & \text{if } trem(x, y) + 1 \neq x \end{cases}$$

which gives the following recursion equations:

$$\begin{array}{lcl} trem(x, 0) & = & 0 \\ trem(x, y+1) & = & mult(trem(x, y) + 1, sg(|x - (trem(x, y) + 1)|)) \end{array}$$

from which we get $\boxed{trem = [0 \ \mathbf{pr} \ [mult \circ (s \circ \pi_3^3, [sg \circ [sdiff \circ (\pi_1^3, [s \circ \pi_3^3])]])]]}$ $\dashv$

15. $\underline{tquot(x, y)} = \begin{cases} 0 & \text{if } x = 0 \\ y \textbf{ div } x & \text{otherwise} \end{cases}$

$\vdash$ We have

$$tquot(x, y+1) = \begin{cases} tquot(x, y) + 1 & \text{if } trem(x, y) + 1 = x \\ tquot(x, y) & \text{if } trem(x, y) + 1 \neq x \end{cases}$$

from which we get the following recursion equations.

$$\begin{array}{lcl} tquot(x, 0) & = & 0 \\ tquot(x, y+1) & = & add(tquot(x, y), \overline{sg}(|x - (trem(x, y) + 1)|)) \end{array}$$

Hence $\boxed{tquot = [0 \textbf{ pr } [add \circ (\pi_3^3, [\overline{sg} \circ [sdiff \circ (\pi_1^3, [s \circ \pi_3^3])]])]]}$ $\dashv$

$\square$

The definitions of $trem$ and $tquot$ are total and have been chosen so that for any two numbers a, b, $a = b \times tquot(b, a) + trem(b, a)$ with $trem(b, a) \leq a$.

We may use part 1 of theorem 3.11 to obtain the following corollary, which extends the use of lemma 3.10 to instances of functions. For example the doubling function $double(x)$ (which was previously considered an example of identification of arguments of the addition operation) could be regarded as an **instance** of the binary multiplication operation in which one argument is the constant 2. Similarly the function $f(n) = 2^n$ may be considered an instance of the function $texp$ in which the first argument is the constant 2.

Corollary 3.12 *If $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is primitive recursive then so is any instance of it.*

Proof: By lemma 3.10 we may reorder the arguments of f so that the argument instantiated is the first one. Hence without loss of generality we assume that the first argument of f is *instantiated* to obtain the k-ary function $g(\overrightarrow{y}^k) = f(m, \overrightarrow{y}^k)$ where m is some fixed natural number m. Then by lemma 3.10 we may treat m as a unary function defined as $m^{(1)}(x) = m$. Then clearly $g = [f \circ (m^{(1)}, \pi_2^{k+1}, \cdots, \pi_{k+1}^{k+1})]$. $\square$

Claim 1. Let $f : \mathbb{N} \to \mathbb{N}$ be primitive recursive, then $f^n(x)$, the n-fold application of f is also primitive recursive.

$\vdash$ Define the n-fold application of f to be the function h by the recursion equations.

$$\begin{array}{lcl} h(x, 0) & = & x \\ h(x, t+1) & = & f(h(x, t)) = [f \circ \pi_3^3](x, t, h(x, t))) \end{array}$$

Hence $h = [id \textbf{ pr } [f \circ \pi_3^3]]$ is a primitive recursive function.

$\dashv$

3.3 Primitive Recursive Predicates

We have seen several functions that are primitive recursive. We now turn to the question of designing some primitive recursive predicates on numbers. Note that every predicate yields a value of 0 or 1 depending upon whether it is true or false respectively. In this sense we have already defined two such functions $sg(x)$ and $\overline{sg}(x)$ in theorem 3.11, both of which could be interpreted as predicates standing for the properties "*is-non-zero*(x)" and "*is-zero*(x)" respectively.

Theorem 3.13 *The following underlined predicates are primitive recursive and hence are also URM-computable.*

Proof outline: Unlike the case of theorem 3.11 we do not give complete expressions and leave the formulation of the actual definition to the reader. We explain only the key ideas that may be used in formulating the primitive recursive definition.

1. the equality relation.

   $\vdash \chi_=(x, y) = \overline{sg}(sdiff(x, y))$ $\dashv$

2. the less-than relation

   $\vdash \chi_<(x, y) = sg(monus(x, y))$

   $\dashv$

3. the greater-than relation.

   $\vdash \chi_>(x, y) = \chi_<(y, x)$

   $\dashv$

4. the "divisor-of" relation.

   $\vdash$ Adopting the convention that 0 is a divisor of 0, but 0 is not a divisor of any positive integer, we get $\chi_|(x, y) = \overline{sg}(trem(x, y))$ $\dashv$

□

Often in the interest of readability we will use the relational symbols, "=", "<", ">" and "|" for the obvious relations they denote, instead of their characteristic functions.

3.4 Some Forms of Primitive Recursion

Corollary 3.14 (Definition by cases). *Let $f_1, f_2, \ldots, f_m : \mathbb{N}^k \nrightarrow \mathbb{N}$ be URM-computable functions and $p_1, p_2, \ldots, p_m : \mathbb{N}^k \to \{0,1\}$ be URM-decidable predicates such that for any $\overrightarrow{x} \in \mathbb{N}^k$ exactly one of $p_1(\overrightarrow{x}), \cdots, p_m(\overrightarrow{x})$ is true. Then the function $g : \mathbb{N}^k \nrightarrow \mathbb{N}$*

$$g(\overrightarrow{x}) = \begin{cases} f_1(\overrightarrow{x}) & if \quad p_1(\overrightarrow{x}) \\ \vdots & \\ f_m(\overrightarrow{x}) & if \quad p_m(\overrightarrow{x}) \end{cases}$$

is URM-computable. Further if $f_1, f_2, \ldots, f_m, p_1, p_2, \ldots, p_m$ are each primitive recursive then so is g.

Proof: $g(\overrightarrow{x}) = \chi_{p_1}(\overrightarrow{x}).f_1(\overrightarrow{x}) + \cdots + \chi_{p_m}(\overrightarrow{x}).f_m(\overrightarrow{x})$. □

Corollary 3.15 (Algebra of Decidability). *If $p, q : \mathbb{N}^k \to \{0,1\}$ are URM-decidable predicates then so are the following*
(i) $\neg p$ *(ii) $p \wedge q$* *(ii) $p \vee q$*
Further if p and q are primitive recursive then so are $\neg p$, $p \wedge q$ and $p \vee q$.

Proof:
(i) $\chi_{\neg p}(\overrightarrow{x}) = monus(1, \chi_p(\overrightarrow{x}))$
(ii) $\chi_{p\wedge q}(\overrightarrow{x}) = \chi_p(\overrightarrow{x}).\chi_q(\overrightarrow{x})$
(iii) $\chi_{p\vee q}(\overrightarrow{x}) = max(\chi_p(\overrightarrow{x}), \chi_q(\overrightarrow{x}))$ □

Bounded sums and Products

Theorem 3.16 *Let $f, g, h : \mathbb{N}^{k+1} \nrightarrow \mathbb{N}$ such that*

$$g(\overrightarrow{x}, y) = \sum_{z<y} f(\overrightarrow{x}, z) = \begin{cases} 0 \quad if \quad y = 0 \\ \sum_{z<w} f(\overrightarrow{x}, z) + f(\overrightarrow{x}, w) \quad if \quad y = w+1 \end{cases}$$

$$h(\overrightarrow{x}, y) = \prod_{z<y} f(\overrightarrow{x}, z) = \begin{cases} 1 \quad if \quad y = 0 \\ \prod_{z<w} f(\overrightarrow{x}, z).f(\overrightarrow{x}, w) \quad if \quad y = w+1 \end{cases}$$

Then g and h are URM-computable if f is total and URM-computable and g and h are primitive recursive if f is primitive recursive.

Proof: Since the above are definitions by primitive recursion it follows that g and h are computable (respectively primitive recursive). More rigorously we have

$$\boxed{g = [0\ \mathbf{pr}\ [add \circ (\pi^{k+2}_{k+2}, [f \circ (\pi^{k+2}_1, \cdots, \pi^{k+1}_{k+1})])]]}$$

$$\boxed{h = [1\ \mathbf{pr}\ [mult \circ (\pi^{k+2}_{k+2}, [f \circ (\pi^{k+2}_1, \cdots, \pi^{k+1}_{k+1})])]]}$$

□

Corollary 3.17 *If $f, i : \mathbb{N}^{k+1} \to \mathbb{N}$ are total computable functions then so are $g(\overrightarrow{x}, y) = \sum_{z<i(\overrightarrow{x},y)} f(\overrightarrow{x}, z)$ and $h(\overrightarrow{x}, y) = \prod_{z<i(\overrightarrow{x},y)} f(\overrightarrow{x}, y)$ Further if f and i are primitive recursive then so are g and h.*

Bounded Quantification

Theorem 3.18 *Let $p(\overrightarrow{x}, y)$ be any predicate and let $q(\overrightarrow{x}, y) = \forall z < y[p(\overrightarrow{x}, z)]$ and $r(\overrightarrow{x}, y) = \exists z < y[p(\overrightarrow{x}, z)]$.*

1. *If p is URM-computable then so are q and r.*
2. *If p is primitive recursive then so are q and r.*

Proof: We have $\chi_q(\overrightarrow{x}, y) = \prod_{0 \le z < y} \chi_p(\overrightarrow{x}, z)$ and $r(\overrightarrow{x}, y) = \neg(\forall z < y[\neg p(\overrightarrow{x}, z)])$ from which the relevant results follow. □

Bounded Minimalization

Let $p : \mathbb{N}^{k+1} \to \{0, 1\}$ be a $\overbrace{(total)\ computable}^{decidable}$ predicate. Then the function

$$g(\overrightarrow{x}, y) = \mu z < y[p(\overrightarrow{x}, z)] = \begin{cases} \text{least } z < y : p(\overrightarrow{x}, z) \text{ if } z \text{ exists} \\ y \text{ otherwise} \end{cases} \tag{3.3}$$

That is, $g(\overrightarrow{x}, y) = z_0 \Leftrightarrow \forall z[0 \le z < z_\circ \le y \Rightarrow \neg p(\overrightarrow{x}, z)]$

Theorem 3.19 *g in (3.3) above is computable if p is URM-computable. Further if p is primitive recursive then so is g.*

Proof: Consider all the predicates $\neg p(\overrightarrow{x}, z)$ for each value of z. For each $0 < v < y$ we have

$$f(\overrightarrow{x}, v) = \prod_{u \leq v} \neg p(\overrightarrow{x}, u) = \begin{cases} 1 & \text{if } \forall u : 0 \leq u \leq v : \neg p(\overrightarrow{x}, u) = 1 \\ 0 & otherwise \end{cases}$$

For each $0 \leq w < y$ we have

$$\sum_{0 \leq w < y} f(\overrightarrow{x}, w) = \begin{cases} w & \text{if } w < z_0 \\ z_0 & \text{if } w > z_0 \end{cases}$$

Hence $g(\overrightarrow{x}, y) = \sum_{w<y} \left(\prod_{v \leq w} \neg p(\overrightarrow{x}, v) \right)$ which is a primitive recursive function of p. □

Corollary 3.20 *If $p : \mathbb{N}^{k+1} \rightarrow \{0, 1\}$ is decidable and $f : \mathbb{N}^{k+m} \rightarrow \mathbb{N}$ is a total computable function then $g(\overrightarrow{x}, y) = \mu z < f(\overrightarrow{x}, \overrightarrow{w})[p(\overrightarrow{x}, z)]$ is a total computable function.*

Theorem 3.21 *The following functions are primitive recursive.*

1. *$\#div(x)$ = the number of divisors of x (assuming $\#div(0) = 1$)*
2. *$prime(x)$ = "whether x is a prime"*
3. *p_x = the x-th prime number (assuming $p_0 = 0$)*
4. *$(x)_y$ = the exponent of p_y in the prime factorization of x.*

Proof:

1. $\#div(x) = \sum_{y \leq x} y \mid x$
2. $prime(x) = eq(\#div(x), 2)$
3. $p_0 = 0$ and $p_{y+1} = \mu z \leq (p_y! + 1)[(p_y < z) \wedge prime(z)]$
4. $(x)_y = \mu z < x[p_y^{z+1} \nmid x]$

□

Theorem 3.22 *The set $PRIMREC$ of primitive recursive functions is countably infinite.*

Proof: It suffices to prove the following two claims.

Claim 1. The set $PRIMREC$ is not finite.

$\vdash$ By contradication . Assume there are only finitely many primtive recursive functions of arity n for some $n \geq 0$, say $f_1^{(n)}, \cdots, f_m :^{(n)}$. Since they are all total we may, construct a new total function $g^{(n)} : \mathbb{N}^n \to \mathbb{N}$ such that $g^{(n)}(\overrightarrow{x}) = max(f_1^{(n)}(\overrightarrow{x}), \cdots, f_m^{(n)}(\overrightarrow{x}) + 1$. $g^{(n)}$ is different from every one of $f_1^{(n), \ldots, f_m^{(n)}}$ and since max is primitive recursive and $+$ is primitive recursive it follows that $g^{(n)}$ is primitive recursive contradicting the initial assumption. Hence $PRIMREC^{(n)}$ is infinite. $\dashv$

Claim 2. The set $PRIMREC$ is countably infinite

$\vdash$ From definition 3.3 it is clear that every primitive recursive function belongs $PRIMREC_j$ for some $j \geq 0$. We may define the *depth* of a primitive recursive function f to be the smallest index j, such that $f \in PRIMREC_j$. For each $j \geq 0$, let PR_j denote the set of primitive recursive functions of depth j. Clearly then, we have $\bigcup_{j \geq 0} PR_j = PRIMREC$ and for each $i \neq j$, $PR_i \cap PR_j = \emptyset$. In particular, we have $PR_0 = PRIMREC_0$ which is countably infinite and by induction we may show that each PR_j is also at most countably infinite. Since $PRIMREC$ is the countable union of at most countable sets, $PRIMREC$ is countably infinite. $\dashv$

$\square$

Chapter 4

The Partial Recursive Functions

Unbounded Minimization

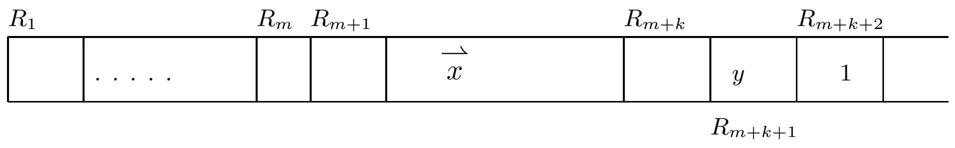
Let $p : \mathbb{N}^{k+1} \longrightarrow \{0,1\}$ be a decidable predicate. Then $g : \mathbb{N}^k {+}\!\!\!\!\!\longrightarrow \mathbb{N}$ defined by

$g(\overrightarrow{x}) \triangleq \mu[p(\overrightarrow{x}\ .y)]$ is computable where

$$= \begin{cases} \text{least } y : p(\overrightarrow{x}, y) \ \ i.e\ \ (\forall z < z_\circ[\neg p(\overrightarrow{x}, z)]) \wedge p(\overrightarrow{x}, z_\circ) \\ \perp \ \text{otherwise} \qquad\qquad \Leftrightarrow g(\overrightarrow{x}) = z_\circ \end{cases}$$

Theorem 4.1 *Unbounded Minimizationis URM-computable.*

Proof: Let g and p be as defined above and let P be a URM program that computes p. Let $\quad m = max(k+1, \rho(p)))$. Then



$\square$

1. $C(1, m+1)$
$\vdots$
k. $C(k, m+k)$ $\longrightarrow$ Copy $\overrightarrow{x}$ to R_{m+1} to R_{m+k}

$k+1$. $Z(m, k+k)$ $\longrightarrow$ Initialize R_{m+k+1} to 0.

$k+2$. $S(m, k+2)$ $\longrightarrow$ Initialize R_{m+k+2} to 1 for comparision

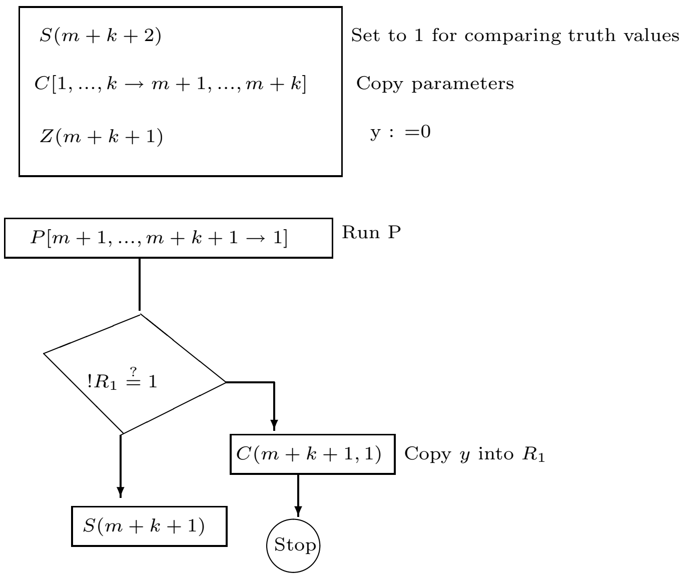
$k+3$. $P(m+1, ..., m+k+2 \to 1)$ $\longrightarrow$ run P and have output in R_1

$k+4$. $J(1, m+k+2, k+7)$ $\longrightarrow$ compare R_1 and R_{m+k+2} and jump

$k+5$. $S(m+k+1)$ $\longrightarrow$ otherwise increment R_{m+k+1}

$k+6$. $J(1, 1, k+3)$

$k+7$. $C(m+k+1, 1)$



Partial Recursive Function

Definition 4.1 *The class PRIMREC of primitive recursive functions is the smallest class*

(i) Containing the basic functions ———————— (o, s, id_i^k)

(ii) Closed under composition and ———————— $\circ[f; g_1, ..., g_k]$

(iii) Closed under primitive recursion —————— $Pr[f, g]$

Definition 4.2 *The class PARTREC of partial recursive functions is the smallest class*

(i) Containing the basic functions ———————— (o, s, id_i^k)

(ii) Closed under composition and ———————— $\circ[f; g_1, ..., g_k]$

(iii) Closed under primitive recursion ——— $Pr[f, g]$

(iv) Closed under (unbounded) minimalization. $\mu y[\rho]$

Theorem 4.2 $\boxed{PRIMREC \subseteq PARTREC \subseteq URM}$

Proof: The foregoing results have shown this. □

Theorem 4.3 *Every URM-computable function is partial recursive.*

Proof: Let $f : \mathbb{N}^k \text{+}\!\!\longrightarrow \mathbb{N}$ be a URM-computable function implemented by a program $P = I_1, \ldots, I_s$. A step of the execution of P is the execution of a single instruction.

Now consider the following functions

$$c_1(\overrightarrow{x}, t) \triangleq \begin{cases} !R_1 \text{ if } P \text{ has stopped executing in less than } t \text{ steps} \\ !R_1 \text{ after t-steps of execution of } P \text{ otherwise} \end{cases}$$

$$j(\overrightarrow{x}, t) \triangleq \begin{cases} 0 \text{ if } P \text{ has stopped execution in less than } t\text{- steps} \\ \text{the number of the next instruction to be executed after } t\text{-steps otherwise} \end{cases}$$

Claim 1. $c_1, \jmath : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ are total fuctions.

Claim 2. $c_!, \jmath$ are both URM-computable. In fact c_1 and $\jmath$ are both primitive recursive.

Claim 3. f is a partial recursive function

If for any $\overrightarrow{x}, \mathbb{N}^k, f(\overrightarrow{x}) \in \mathbb{N}$ then P converges after some t_0 steps. where $t_0 = \mu t[\jmath(\overrightarrow{x}, t) = 0]$ and R_1 contains the value $f(\overrightarrow{x})$. Hence $f(\overrightarrow{x}) = c_1(\overrightarrow{x}, t_0) = c_1(\overrightarrow{x}, \mu t[j(\overrightarrow{x}, t) = 0])$ □

Corollary 4.4 *A function is URM-computable if and only if it is partial recursive. That is,*

$$\boxed{PARTREC \subseteq URM}.$$

Chapter 5

Coding and Gödelization

5.1 Introduction

We have discussed the notion of

5.2 Numbering and Coding

Definition 5.1 *A set A is* **denumerable** *(or* **countably infinite***) if there exists a bijection $f : A \longrightarrow \mathbb{N}$.*

Definition 5.2 *An* **enumeration with possible repetitions** *of a set A is a surjective function $g : \mathbb{N} \longrightarrow A$, and if g is injective too (i.e. g is a bijection) then g is an* **enumeration without repetitions***. Clearly g^{-1} shows that the set is denumerable.*

Definition 5.3 *A set A is* **effectively denumerable** *if there is a bijection $f : X \xrightarrow[onto]{1-1} \mathbb{N}$ such that both f and f^{-1} are effectively computable functions.*

Theorem 5.1 *The following sets are effectively denumerable*

1. $\mathbb{N} \times \mathbb{N}$.
2. $\mathbb{N}^k$ *for all* $k > 2$.
3. $\mathbb{N}^*$.

Proof:

1. $pair : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ with $\boxed{pair(m, n) \triangleq 2^m(2n + 1) - 1)}$. $pair$ is primitive recursive and hence effective. The inverse functions $fst2, snd2 : \mathbb{N} \rightarrow \mathbb{N}$ which satisfy the equations
$$pair(fst2(z), snd2(z)) = z \tag{5.1}$$
$$fst2(pair(x, y)) = x \tag{5.2}$$
$$snd2(pair(x, y)) = y \tag{5.3}$$
are defined as follows. $fst2(z)$ for any natural z is the exponent of the largest power of 2 that divides $z + 1$, which may be equivalently expressed as 1 less than the smallest power of 2 that does not divide $z + 1$. Hence $fst2(z) = \mu x < (z + 2)[2^x \nmid (z + 1)] - 1$ which is clearly primitive recursive. $snd2(z)$ is simply the largest odd divisor of $z + 1$. Equivalently, $snd2(z) = tquot(fst2(z), z + 1)$ which again is primitive recursive.

2. We prove this only for $k = 3$. $triple(x, y, z) = pair(x, pair(y, z))$ and its inverses $fst3(u) = fst2(u)$, $snd3(u) = fst2(snd2(u))$ and $thd3(u) = snd2(snd2(u))$. It is clear that all these functions are primitive recursive and they satisfy the following defining equations.
$$triple(fst3(u), snd3(u), thd3(u)) = u \tag{5.4}$$
$$fst3(triple(x, y, z)) = x \tag{5.5}$$
$$snd3(triple(x, y, z)) = y \tag{5.6}$$
$$thd3(triple(x, y, z)) = z \tag{5.7}$$

3. Consider the function $seq : \mathbb{N}^* \rightarrow \mathbb{N}$, defined as
$$\begin{array}{rcl} seq([a_1, \ldots, a_k]) & \triangleq & 2^{a_1} + 2^{a_1+a_2+1} + \cdots + 2^{a_1+a_2+a_k+(k-1)} \\ & = & 2^{a_1}(1 + 2^{a_2+1}(1 + 2^{a_3+1}(\ldots + 2^{a_k+1})\ldots))) \end{array}$$
For the empty sequence ε, we have $seq(\varepsilon) = 0$, whereas for all other sequences $s \in \mathbb{N}$, $seq(s) > 0$. For instance, the sequence $seq([0]) = 1$, $seq([1]) = 2$, and $seq([0, 0]) = 2^0(1 + 2^{(0+1)}) = 3$. We define the function $hd(z)$ as the largest power of 2 that divides z i.e. $hd(z) = \mu x < (z + 1)[2^x \nmid (z + 1)] - 1$ and $tl(z) = tquot(2, tquot(hd(z), z) - 1)$ to ensure that if $z = seq([a_1, a_2, \ldots, a_k])$ then $tl(z) = seq([a_2, \ldots, a_k])$. We also define the function $cons(a, z) = 2^a(2z + 1)$. It is then clear that the functions seq, hd, and tl satisfy the following equations.
$$seq(s) = 0 \text{ iff } s = \varepsilon \tag{5.8}$$
and whenever $s = [a_1, \ldots, a_k]$ for $k > 0$, we have
$$hd(cons(a, z)) = a \tag{5.9}$$
$$tl(cons(a, z)) = z \tag{5.10}$$
$$seq([a_1, \ldots, a_k]) = cons(a_1, seq([a_2, \ldots, a_k])) \tag{5.11}$$

□

5.3 Gödel Numbers of URM Instructions

Theorem 5.2 *The set of instructions of the URM is effectively denumerable.*

Proof:

We use the opcodes of the instructions to code them up modulo 4. Consider the function β such that

$$\begin{array}{lcl} \beta(\mathsf{Z}(n)) & = & 4(n-1)+0 \\ \beta(\mathsf{S}(n)) & = & 4(n-1)+1 \\ \beta(\mathsf{C}(m,n)) & = & 4.pair(m-1,n-1)+2 \\ \beta(\mathsf{J}(m,n,p)) & = & 4.triple(m-1,n-1,p-1)+3 \end{array}$$

Hence every instruction I has a unique code given by $\beta(I)$. Conversely, for any $x \in \mathbb{N}$, it is possible to decode the instruction I primitive recursively by first determining the opcode (x $\underline{mod}$ 4) and then depending upon the opcode it is possible to determine the parameters. These operations are all primitive recursive. □

5.4 Gödel Numbering of Programs

Theorem 5.3 *$\mathscr{P}$=collection of all URM programs is effectively denumerable.*

Proof: Any program $P = I_1, ..., I_k$ has a Gödel number given by $seq[\beta(I_1), \ldots, \beta(I_k)]$, where all the functions used are bijective and primitive recursive, includeing $|P| = k$ which can be determined as follows.

Let $i = \mu x[2^{x+1} \nmid P + 1]$ then

$i = a_1 + a_2 + ...a_k + k$.

Compute each of the indices and then effectively compute k. □

Notation. Let $PARTREC$ be partitioned by arity

$$\boxed{\begin{array}{l} PARTREC = \bigcup_{n>0} PARTREC^{(n)} \\ \text{where each } f^{(n)} \in PARTREC^{(n)} : \mathbb{N}^n \rightharpoonup \mathbb{N}. \\ \text{We use the superscript } ^{(n)} \text{ denote arity.} \end{array}}$$

Since γ is obtained only through the composition of bijective functions(which have inverses) we have γ^{-1} also exists and is primitive recursive

Theorem 5.4 *The set $PARTREC^{(n)}$*[1]

Proof: Consider the enumeration

$Q_0^{(n)}\ , Q_1^{(n)}\ , Q_2^{(n)}\ , ...$

with repetitions. We now construct an enumeration without repetitions as follows.

$f(0) = 0$

$$\begin{aligned} f(m+1) = \mu z[& Q_z^{(n)} \neq Q_{f(0)}^{(n)}\ \wedge \\ & Q_z^{(n)} \neq Q_{f(1)}^{(n)}\ \wedge \\ & \quad \vdots \\ & Q_z^{(n)} \neq Q_{f(m)}^{(n)} \\ &] \end{aligned}$$

which clearly exists as a function and provides an enumeration since

$f(m+1)$ is a function(different from each of $f(0), ...f(m)$)

$\square$

[1] $PARTREC^{(n)}$ is clearly infinite since $PRIMREC^{(n)} \subseteq PARTREC^{(n)}$ is infinite

Chapter 6

The Hierarchy of Primitive Recursive Functions

6.1 Introduction

So far we have seen that the URM has the power of partial recursive functions. We have already seen that the primitive recursive functions are total. In many cases such as subtraction and division we have made them total by adopting conventions which at best may be considered unconventional. We leave it to the reader as an exercise to define partial recursive versions of functions like subtraction and division which conform to normal mathematical convention (e.g. division by 0 should be undefined, subtraction of a larger number from a smaller number should be undefined etc.).

This might give the reader the feeling that the difference between the primitive recursive functions and the URM might only be in the question of being able to program undefinedness by non-terminating programs.

Two questions naturally arise from the material presented so far.

1. Does unbounded minimalization yield genuine computational power that goes beyond primitive recursion. Put differently,

   $\boxed{\textit{Are there partial recursive functions that are } \text{total} \textit{ but not primitive recursive?}}$

   If so, then unbounded minimalization does yield genuine computational power.

2. If the answer to the above question is negative, then we might ask whether it is possible to get rid of undefinedness completely by eliminating non-termination from URM programs. In other words,

   $\boxed{\textit{Is there a machine that exactly characterises the primitive recursive functions?}}$

The answer to this question has a direct bearing on the relative power of machines and it might enable us to construct and program machines which yield only total functions.

In the following sections we answer both these questions.

6.2 URM–: A Restricted Machine for Primitive Recursion

While we have shown that primitive recursion is URM-computable, we have not explored the possibility of restricting the instruction set of the URM so as to characterize the primitive recursive functions.

One of the instructions of the URM machine which makes it powerful in many ways including that of leading to non-termination is the jump instruction $\mathsf{J}(m, n, p)$. In fact, in the 1970s, it was recognized that such arbitrary transfers of control in any programming language led to serious difficulty in "debugging" faulty programs, made all programs (except the trivial ones) completely incomprehensible even to the authors of the programs and more importantly made reasoning about the behaviours of programs an almost impossible task. In fact, short of actually manually executing the programs it was impossible to make any except the most trite statements about what such programs do.

In this section we introduce a more structured construct for URM programs that directly reflects the the nature of primitive recursion. Let the new machine be called $URM-$. The instruction set of this machine is summarised in the table below.

opcode	instruction	semantics	Verbal description
0	$\mathsf{Z}(n)$	$R_n := 0$	Clear register R_n
1	$\mathsf{S}(n)$	$R_n :=!R_n + 1$	Increment the contents of register R_n
2	$\mathsf{C}(m, n)$	$R_n :=!R_m$	Copy the contents of register R_m into R_n
3	$\mathsf{L}(m)$		Execute $!R_m$ times the following instructions upto the matching E.
4	E		End of the code to be repeated

The two instructions $\mathsf{L}(m)$ and E (which stand for "Loop" and "End") satisfy the following conditions

1. Each occurrence of the $\mathsf{L}(\ldots)$ instruction in a program is *followed* by a matching occurrence of E.
2. The sequence of instructions between $\mathsf{L}(m)$ and E (called the **loop body**) is executed $!R_m$ times. The sequence is entirely skipped if $R_m = 0$.

3. The two instructions $\mathsf{L}(m)$ and E along with the loop body form the **loop**, R_m the **loop counter** and the contents of R_m is the **loop bound**.
4. There can be no instruction in the loop body which modifes the loop bound.
5. Loops may be *nested* one within the other i.e. the loop body of one loop may itself contain a loop. But in the case of nested loops, all the loop counters must be distinct.
6. For any two loops, either the two loops are entirely disjoint or one is entirely contained in the other.

Notes.

1. Notice that condition 1. clearly prohibits the E instruction preceding any $\mathsf{L}(\ldots)$ instruction.
2. The reader is encouraged to ponder over why we have put in the condition 4. On the other hand,
3. condition 6. is redundant if we follow the convention that each E instruction matches the closest $\mathsf{L}(\ldots)$ instruction that *precedes* it.
4. Unlike the case of the URM, instruction numbers are needed only for the purpose of maintaing the program counter in the execution of the machine. Since there are no jump instructions, instruction numbers are unnecessary for the purpose of understanding and reasoning about the program.

Definition 6.1 *The class of loop-free $URM-$ programs is $LOOP_0$. $LOOP_1$ is the class of programs containing $LOOP_0$ and all programs with loops such that there are no loops in any of the loop bodies i.e. $LOOP_1$ is the class of $URM-$ programs with loops nested at most 1 deep. For any $n > 0$, the class $LOOP_n$ consists of $LOOP_{n-1}$ and all those programs which contain loops nested at most n-deep. $LOOP = \bigcup_{n\geq 0} LOOP_n$ is the class of all $URM-$ programs.*

Theorem 6.1 *Every primitive recursive function can be implemented as a $URM-$ program.*

Proof: It is clear that the initial functions are all in $LOOP_0$. Hence it suffices to show that generalized composition and primitive recursion may be programmed in $URM-$. The case of generalized composition also follows the proof given for the URM. That leaves only primitive recursion. Assume $h = [f\ \mathbf{pr}\ g]$. Assuming that F and G are respectively programs that implement f and g respectively, we proceed almost exactly as we did in the URM machine except that the two jump instructions are replaced by $\mathsf{L}(t+2)$ and E respectively. $\square$

Theorem 6.2 *Every program of $URM-$ implements a primitive recursive function.*

Proof outline: First of all notice that each of the following claims is easy to prove and taken together they prove the theorem.

Claim 1. Every $URM-$ program terminates.

Claim 2. If P and Q are $URM-$ programs that implement primitive recursive functions, their join PQ also implements a primitive recursive function.

Claim 3. If F implements a primitive recursive function f and R_m is a register whose contents are not modified by F, then $\mathsf{L}(m), F, \mathsf{E}$ is a program that implements f^r where $!R_m = r$, i.e. the loop construct implements the r-fold application of the body of the loop. $\square$

So we have shown that the primitive recursive functions are exactly characterised by the class of functions that may be implemented in the $URM-$.

In the next section we show that not all total computable functions are primitive recursive.

6.3 A Non-Primitive Recursive Total Partial Recursive function

An example of a function that is partial recursive and total is the *Ackermann function* which is defined as follows.

$$\begin{array}{lcl} Ack(0,y) & = & y+1 \\ Ack(x+1,0) & = & Ack(x,1) \\ Ack(x+1,y+1) & = & Ack(x, Ack(x+1,y)) \end{array}$$

6.3.1 A Variant of the Ackermann function

We consider a variation of the Ackermann function (in the sense that recursion is of the same kind, though the individual cases are different).

$$A(x,0) = 1 \tag{6.1}$$

$$A(0,y) = \begin{cases} y+1 & \text{if } y \leq 1 \\ y+2 & \text{if } y > 1 \end{cases} \tag{6.2}$$

$$A(x+1,y+1) = A(x, A(x+1,y)) \tag{6.3}$$

It is easy to show that the above equations do define a total function and that is partial recursive.

Lemma 6.3 *The equations 6.1-6.3 define a unique total function.*

Proof outline: Define a finite relation $S \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ to be *suitable* if it satisfies the following conditions.

1. $(x, 0, z) \in S$ implies $z = 1$,
2. $(0, y, z) \in S$ implies $z = y + 1$ if $y \leq 1$ and $z = y + 2$ otherwise.
3. $(x, y, z) \in S$ for $x > 0$ and $y > 0$ implies there exists u such that $(x, y - 1, u) \in S$ and $(x - 1, u, z) \in S$.

For each (x, y) let $S_{(x,y)}$ denote the smallest suitable set such that there exists a triple of the form $(x, y, z) \in S_{(x,y)}$,

Claim 1. For each (x, y), $S_{(x,y)}$ is nonempty and contains a unique triple $(x, y, z) \in S_{(x,y)}$.

$\vdash$ Consider the lexicographic ordering $<_{lex}$ on ordered pairs (x, y). It is clear from the definition of suitable sets that for each ordered pair of the form $(x, 0)$, $S_{(x,0)} = \{(x, 0, 1)\}$ and for each ordered pair of the form $(0, y)$, $S_{(0,y)} = \{(0, y, z) \mid z = y + 1 \text{ if } y \leq 1, z = y + 2 \text{ otherwise}\}$. Consider any ordered pair (x, y) with $x > 0$ and $y > 0$. Assume that for each $(x', y') <_{lex} (x, y)$ there exists $S_{(x',y')} \neq \emptyset$ and such that there is a unique triple $(x', y', z') \in S_{(x',y')}$. Then $S_{(x,y)} = S_{(x,y-1)} \cup S_{(x-1,u)} \cup \{(x, y, z)\}$ where $u = S_{(x,y-1)}(x, y - 1)$ and $z = S_{(x-1,u)}(x - 1, u)$. Clearly then $S_{(x,y)}(x, y) = z$ uniquely defines $S_{(x,y)}$ as the smallest suitable set containing a unique triple (x, y, z). $\dashv$

□

Lemma 6.4 *A is a partial recursive function.*

Proof: Consider the triples $(x, y, z) \in A$. We may code the triples using some coding mechanism such as $triple1(x, y, z) = pair1(x, pair1(y, z))$, where $pair1(x, y) = 2^x(2y + 1)$.

Claim 1.

1. $triple1$ is a primitive recursive function.
2. $triple1$ is a bijective function from $\mathbb{N}^3$ to $\mathbb{N}$.
3. The decoding functions fst, snd, thd which yield respectively, the first, second and third components of a triple are also primitive recursive.
4. For any $a, b, c, d \in \mathbb{N}$, the following holds

$$\boxed{triple1(a, b, c) = d \text{ iff } a = fst(d),\ b = snd(d) \text{ and } c = thd(d)}$$

We may also code finite sets of numbers $T = \{a_1, \ldots, a_m\}$ by the function $fs(T) = \prod_{a_i \in T} p_{a_i+1}$ where p_k denotes the k-th prime number (with $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ etc.) for any positive integer k.

Claim 2. fs is a primitive recursive function

$\vdash$ Since multiplication and determining the k-th prime are both primitive recursive (the k-th prime can be found by bounded minimalization, since $p_k < 2^{2^k}$ which may be proven from Euclid's proof [1]), it follows that fs is also primitive recursive. $\dashv$

Claim 3. Given any positive integer, the predicate $isfs : \mathbb{N} \to \{0,1\}$ which determines whether a given number is the code of a finite set, is primitive recursive.

$\vdash$ Just check that the number is square-free. $\dashv$

Claim 4. Any finite set $T = \{(a_i, b_i, c_i) \mid 1 \leq i \leq n\}$ of triples of numbers may be encoded as $fs(\{triple1(a_i, b_i, c_i) \mid 1 \leq i \leq n\})$.

Claim 5. For any $(x, y, z) \in \mathbb{N}^3$ and a finite set $T \subseteq \mathbb{N}$ we have

$$\boxed{(x, y, z) \in T \text{ iff } divisorof(p_{triple1(x,y,z)}, fs(T))}.$$

Claim 6. For any $v \in \mathbb{N}$, the predicate $isSuitable(v)$ (which determines whether v is the code of a suitable set) is primitive recursive.

$\vdash$ We outline a method of checking that is intuitively primitive recursive, because every search is bounded by the number v. For each prime divisor of v, obtain the triple and check for membership of the "earlier" triples that may be needed to compute it. Given any triple (x, y, z) in the set, if $x = 0$ then it is only necessary to verify that $z = y + 1$ or $z = y + 2$ according as whether $y \leq 1$ or $y > 1$. Similarly, if $y = 0$, then it is only necessary to verify that $z = 1$. If on the other hand both $x > 0$ and $y > 0$ then it is clearly necessary to check for the presence of "earlier" triples. An earlier triple in this case is one of the form $(x, y-1, u)$ where u is unknown and needs to be determined. But having determined u (since u is unique!), it is only necessary to verify that the triple $(x-1, u, z)$ belongs to the set (which merely involves checking whether $p_{triple1(x-1,u,z)}$ is a divisor of v. However, if $y > 1$ then this process needs to be pursued with $(x, y-1, u)$. Similarly if $x > 1$, then a similar process needs to be pursued with $(x-1, u, z)$. But

[1]By Euclid's proof of the infinitude of primes, it follows that $p_k \leq 1 + \prod_{i=1}^{k-1} p_i$. Using this very fact and starting from $2 < 2^{2^1}$ we may prove by induction that $p_i < 2^{2^i}$ for each $i < k$ and then conclude that $p_k < 1 + \prod_{i=1}^{k-1} 2^{2^i} = 1 + 2^{2^1 + \cdots + 2^{k-1}} \leq 2^{2^k}$.

each of these processes is bounded by the value v and the checking involves merely decoding and checking for prime divisors. Hence these processes are all bounded and intuitively primitive recursive. $\dashv$

Claim 7. The predicate $R(x,y,v)$ defined as $isSuitable(v) \wedge \exists z < v[(x,y,z) \in S_v]$ is primitive recursive.

Claim 8. $A(x,y) = \mu z[(x,y,z) \in S_{f(x,y)}]$ where $f(x,y) = \mu v[R(x,y,v)]$.

$\vdash$ Firstly of all the suitable sets that contain a triple (x,y,z), $S_{(x,y)}$ is the smallest and is contained in every other suitable set which contains the triple. This follows from the construction of the smallest suitable sets required for each pair (x,y). Further our coding mechanisms for sets ensures that $S \subset T$ implies $fs(S) < fs(T)$ and $fs(S)|fs(T)$. Hence $f(x,y)$ yields the code of the set $S_{(x,y)}$ in which there is a unique triple (x,y,z) which satisfies the equations 6.3. $\dashv$

$\square$

We have shown that $A(x,y)$ is total and partial recursive. In order to prove that it is *not* primitive recursive it does not suffice to simply show that it is partial recursive. Many primitive recursive functions may be shown to be partial recursive. For example in each case of bounded minimalization, if the bound is removed we get an equivalent function that "looks" partial recursive, even though it is not. Hence our question of whether there are total partial recursive functions that are not primitive recursive has to be answered in some more convincing manner by showing that the function cannot exist in the class $PRIMREC$ under any circumstances or by showing that the operations that yield primitive recursive functions are inadequate in some convincing way to be used to define the function A.

6.3.2 The Growth of functions

We first define a family $\mathscr{F}$ of (inter-related) functions and then show that the function A may be expressed in terms of this family. The growth properties of A are related to the growth properties of this family of functions and eventually we will show that it is impossible for any primitive recursive function to match the growth of the function A.

Definition 6.2 *Let $\mathscr{F} = \{f_x \mid x \in \mathbb{N}, f_x : \mathbb{N} \to \mathbb{N}\}$ be a family of functions defined by the equations*

$$\begin{aligned} f_0(y) &= \begin{cases} y+1 & \text{if } y \leq 1 \\ y+2 & \text{if } y > 1 \end{cases} \\ f_{x+1}(y) &= f_x^y(1) \end{aligned}$$

Lemma 6.5 $A(x,y) = f_x(y)$

Proof outline: By double induction on x and y. □

Since we are primarily interested in *asymptotic* growth of functions we define the notion of functions being "nearly" equal, a concept that suffices for our purposes.

Definition 6.3 *Let $f, g : \mathbb{N} \to \mathbb{N}$ be total unary functions. g* **majorizes** *f denoted by $f \prec g$ or $g \succ f$, if $f(x) < g(x)$* almost everywhere, *i.e. there exists x_0 such that for all $x > x_0$, $f(x) < g(x)$. f* **asymptotically approximates** *g, denoted $f \asymp g$, if $f(x) = g(x)$* almost everywhere. *Given a class of functions $\mathscr{F}$, $f \in_{\asymp} \mathscr{F}$ if there exists a function $g \in \mathscr{F}$ such that $f \asymp g$.*

Theorem 6.6 *The following claims on the properties of f_x hold.*

Proof outline:

Claim 1. The first few functions in $\mathscr{F}$ are as follows.

1. $f_1(y) = 2y$ for all $y > 0$.
2. $f_2(y) = 2^y$ for all $y > 0$.
3. $f_3(y) = \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}$ y-times for all $y > 0$.

Claim 2. For all $n > 0$, $f_n \in_{\asymp} LOOP_n$, where $LOOP_n$ is the class of URM– programs with depth of nesting at most n.

$\vdash$ By induction on n. Starting with $n = 1$ we have $f_1(y) \asymp 2y$ which is in turn implemented by the following sequence of instructions $\mathsf{C}(1,3), \mathsf{C}(1,2), \mathsf{L}(2), \mathsf{S}(3), \mathsf{E}, \mathsf{C}(3,1)$. Assume for some $k \geq 1$, $f_k \in_{\asymp} LOOP_k$. That is there exists program $F_k \in LOOP_k$ such that $f_k \asymp \phi_{F_k}$. Let $\rho(F_k) = m_k$ and let $F'_k = F_k[l_1, \ldots, l_{m_k} \to l_1]$, where $l_1, \ldots, l_{m_k}$ are chosen so as not to interfere with any of the parameters or intermediate registers that may be required. We have $f_{k+1}(y) = f_k^y(1)$. Now consider the program F_{k+1} defined by the sequence of instructions $\mathsf{C}(1, m_k + 1), \mathsf{S}(1), \mathsf{L}(m_k + 1), F'_k, \mathsf{E}$. Clearly since $F_k \in LOOP_k$, F_k has a nested loop depth of at most k, F_{k+1} has a depth of at most $k+1$. Further since $f_k \asymp F_k$, we may show that $f_{k+1} \asymp F_{k+1}$. Hence $f_{k+1} \in_{\asymp} LOOP_{k+1}$. $\dashv$

Claim 3. $f_{x+1}(y+1) = f_x^{y+1}(1) = f_x(f_x^y(1)) = f_x(f_{x+1}(y))$.

Claim 4. f_0 is a monotonic function i.e. $y \leq z$ implies $f_0(y) \leq f_0(z)$.

Claim 5. $f_0^k(y) \geq k$

$\vdash$ By induction on k. $f_0^0(y) = y \geq 0$ and $f_0^{k+1}(y) = f_0(f_0^k(y)) \geq f_0^k(y) + 1 \geq k + 1$. $\dashv$

Claim 6. $f_n(y) > y$ i.e. f_n is an increasing function of y.

$\vdash$ By induction on n. For $n = 0$ it is obvious since $f_0(y) \geq y + 1$ always. Assume $f_k(y) > y$ for some $k \geq 0$. Then $f_{k+1}(y) > y$ is proven by induction on y. For $y = 0$ we have $f_{k+1}(0) = f_k^0(1) = 1 > 0 = y$. Assuming $f_{k+1}(m) > m$ i.e $f_{k+1}(m) \geq m + 1$ for some $m \geq 0$ we have $f_{k+1}(m + 1) = f_k(f_{k+1}(m)) > f_{k+1}(m) \geq m + 1$. $\dashv$

Claim 7. $f_n(y + 1) > f_n(y)$ for each n.

$\vdash$ It is easily verified for $n = 0$. Assuming the result is true for some $k \geq 0$, we have $f_{k+1}(y+1) = f_k(f_{k+1}(y)) > f_{k+1}(y)$ $\dashv$

Claim 8. $f_{n+1}(y) \geq f_n(y)$ for all n and x.

$\vdash$ The proof is *not* by induction. However, for $y = 0$ the claim holds, since both sides of the inequality yield 1. For $y > 0$ we have for any n, $f_{n+1}(y+1) = f_n(f_{n+1}(y))$. Since $f_{n+1}(y) > y$ we have $f_{n+1}(y) \geq y + 1$ and further since $f_n(y + 1) > f_n(y)$ we have $f_{n+1}(y + 1) = f_n(f_{n+1}(y)) \geq f_n(y + 1)$. $\dashv$

Claim 9. $f_n^{k+1}(y) > f_n^k(y)$

$\vdash$ $f_n^{k+1}(y) = f_n(f_n^k(y)) > f_n^k(y)$. $\dashv$

Claim 10. $f_n^{k+1}(y) \geq 2f_n^k(y)$

$\vdash$ By induction on k. For $k = 0$ it is clear. For $k > 0$ we have $2f_n^{k+1}(y) = 2f_n(f_n^k(y)) \leq f_n^{k+1}(f_n(y)) = f_n^{k+2}(y)$. $\dashv$

Claim 11. $f_n^{k+1}(y) \geq f_n^k(y) + x$

$\vdash$ By induction on k. $f_n^1(y) \geq 2f_n^0(y) = 2y = f_n^0(y) + y$. Similarly $f_n^{k+1}(y) \geq 2f_n^k(y) = f_n^k(y) + f_n^k(y) \geq f_n^k(y) + y$. $\dashv$

Claim 12. $f_1^k(y) \geq 2^k y$.

$\vdash$ For $k = 0$, $f_1^0(y) = y \geq 2^0.y$ and for $k + 1$ we have $f_1^{k+1}(y) = f_1(f_1^k(y)) = 2f_1^k(y) \geq 2.2^k x = 2^{k+1}x$. $\dashv$

$\square$

Theorem 6.7 *For all n, k, $f_{n+1} \succ f_n^k$.*

Proof outline: By double induction on n, k. For $n = 0$, we have $f_0^k(y) = y + 2k$, for all $y \geq 2$ and $f_1(y) = 2y$. Clearly for all $y > 2k$, we have $2y > y + 2k$. Hence $f_1 \succ f_0^k$. Assume that for some $n > 0$ and all k, $f_n \succ f_{n-1}$. For $k = 0$, we know $f_{n+1}(y) > y = f_n^0(y)$ for all y. Hence $f_{n+1} \succ f_n^0$. Assume $\square$

Corollary 6.8 *For any constant c and all n, k, $f_{n+1} \succ c.f_n^k$.*

Since all URM– programs are terminating, we define a concept of running time of programs.

Definition 6.4 *For any program URM– program P, its* **running time** *on any input $\overrightarrow{x}$ is defined as the total number of executions of* atomic instructions[2] *in P.*

We would like to observe that the running time of any program P may be easily calculated by a URM– program T_P by the following procedure.

- Choose a register R_m with $m > max(k, \rho(P))$, where k si the length of the input.
- To every atomic instruction, add a new instruction $\mathsf{S}(m)$.
- At the end of the program append a new instruction $\mathsf{C}(m, 1)$

Clearly the program T_P for any input $\overrightarrow{x}$ computes the running time of P. Also T_P has the same depth of nesting as P. We then have the following important theorem.

Theorem 6.9 *(***The Bounding theorem***) For all $n \geq 0$ and $P \in LOOP_n$, there exists $k \geq 0$ such that $T_P(\overrightarrow{x}) \leq f_n^k(max(\overrightarrow{x}))$ for every $\overrightarrow{x}$.*

Proof: By induction on n. For $n = 0$, there are no loops in P and hence P consists of only a *constant number* c of atomic instructions. Hence for any input $\overrightarrow{x}$, we have $\phi_{T_P}(\overrightarrow{x}) \leq c \leq f_0^c(max(\overrightarrow{x}))$.

Assume for some $n > 0$, for each $Q \in LOOP_{n-1}$ there exists a constant k_{n-1} such that for all $\overrightarrow{x}$, $T_Q(\overrightarrow{x}) \leq f_{n-1}^{k_{n-1}}(max(\overrightarrow{x}))$. Suppose $P \in LOOP_n$ and let $max(\overrightarrow{x}) = u$. We then have the following cases.

[2] We could equally well have defined the running time as the number of changes in configuration, but as we shall see it is easier to compute running time in terms of the atomic statements executed. Moreover our theory of asymptotic growth of functions does not get seriously affected by either definition.

Case $P \in LOOP_{n-1}$. Then by the induction hypothesis and the previous theorem we have $\phi_{T_P}(\overrightarrow{x}) \leq f_{n-1}^{k_{n-1}}(u) \leq f_n^{k_{n-1}}(u)$.

Case $P \notin LOOP_{n-1}$. We proceed by induction on the structure of P. Since $n > 0$ there exists at least one looping construct in P which is of the form $\mathsf{L}(m), R, \mathsf{E}$, where $R \in LOOP_{n-1}$ and hence $\phi_{T_R}(\overrightarrow{x}) \leq f_{n-1}^{k_R}(u)$ for some constant k_R. For simplicity assume P is made up by joining the programs Q, $\mathsf{L}(m), R, \mathsf{E}$ and S, where Q and S are in $LOOP_{n-1}$. Assume $!R_m = v$ and since Q and S are in $LOOP_{n-1}$, by the induction hypothesis, there exist constants k_Q and k_S such that $\phi_{T_Q}(\overrightarrow{x}) \leq f_{n-1}^{k_Q}(u)$. Further

Claim 1. The value in any register after Q has been executed is bounded above by $u + f_{n-1}^{k_Q}(u)$.

$\vdash$ Since each atomic statement increases the maximum of the values contained in all registers by at most 1, it follows after the execution of Q whose running time is bounded by $f_{n-1}^{k_Q}(u)$, no register can possibly contain a value greater than the sum of its initial value and the running time of Q. $\dashv$

We may argue similarly with the rest of the code in P to conclude that every register has a value bounded above. This gives us the result that

$$\begin{array}{rcl} \phi_{T_P}(\overrightarrow{x}) & \leq & u + f_{n-1}^{k_Q}(u) + v.f_{n-1}^{k_R}(u) + f_{n-1}^{k_S}(u) \\ & \leq & f_{n-1}^{k_Q+1}(u) + v.f_{n-1}^{k_R}(u) + f_{n-1}^{k_S}(u) \\ & \leq & u + (v+2).f_{n-1}^{k_{n-1}}(u) \\ & \prec & f_n^{k_n}(u) \end{array}$$

where $k_{n-1} > max(k_Q + 1, k_R, k_S)$. □

Hence the value in register R_1 in any URM– program can be a priori bound by the sum of the initial value in the register and the running time of the program. This implies that values produced by primitive recursive functions cannot grow faster than the bounds imposed on them by the bounding theorem. However the function $A(x, y)$ grows faster than $f_n^k(1)$ for any fixed n and k. While each of the function f_n is primitive recursive, since $f_n \in LOOP_n$, the function A outstrips them all and hence is not primitive recursive.

Chapter 7

Universality and Parametrisation

Notation

1. P_a denotes the URM program with gödel number a.
2. For any $a \in \mathbb{N}$ and $n \geq 1$

   $\phi_a^{(n)}$ = the n-ary function computed by P_a.

   $= f_{P_a}^{(n)}$

   $D_a^{(n)}$ = the domain of $\phi_a^{(a)}$.

   $R_a^{(n)}$ = the range of $\phi_a^{(n)}$.
3. The above are abbreviated to

   ϕ_a, D_a, R_a respectively when $n = 1$.
4. Since every computable function has an infinite number of programs that implement it, the enumeration

   Q_0, Q_1, Q_2 ,...

   is an enumeration of unary computable functions with repetitions.

Theorem 7.1 *There is a total unary function that is not computable.*

Proof: Assume $Q_0^{(1)}, Q_1^{(1)}, Q_2^{(1)}$,... is an enumeration of

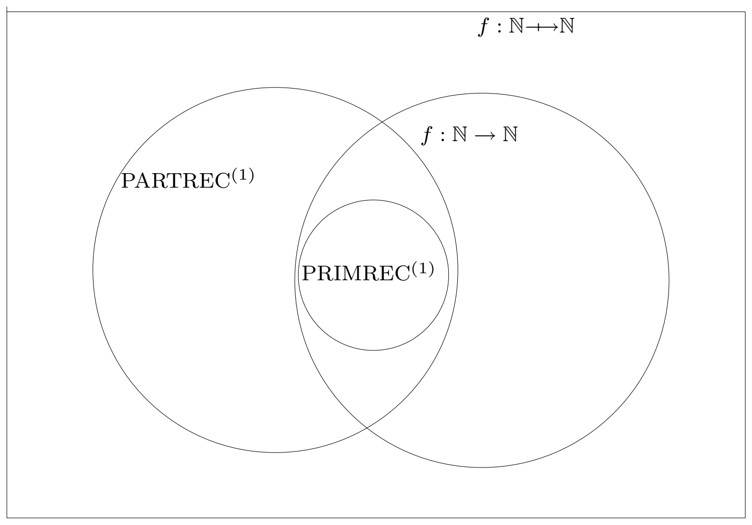
$PARTREC^{(1)}$: Consider the total function $f : \mathbb{N} \to \mathbb{N}$.

$$f(n) = \begin{cases} Q_n^{(1)}(n) + 1 \ if \ \ Q_n^{(1)}(n) \downarrow . \\ 0 \qquad\qquad\quad otherwise. \end{cases}$$

Claim 1. $f(n)$ is the different from every function in the enumeration of computable functions.

Claim 2. $f(n)$ is total. Hence $f(n)$ is not computable □

The following is the picture of computable functions that we have.


$f : \mathbb{N} \rightharpoonup \mathbb{N}$
$f : \mathbb{N} \to \mathbb{N}$
PARTREC$^{(1)}$
PRIMREC$^{(1)}$

7.1 The Parametrisation theorem (Simplified form)

Assume $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ is computable ($\in \ PARTREC^{(2)}$) for each fixed value $a \in \mathbb{N}$. We may define a unary function $g_a : \mathbb{N} \rightharpoonup \mathbb{N}$ such that

$$\forall y \in \mathbb{N} : g_a(y) = f(a, y) \qquad \text{i.e} \qquad g_a \simeq \lambda y[f(a, y)]$$

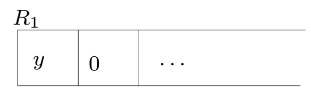
Then clearly since f is computable g_a is also computable and in fact there is a program P_a for each $a \in \mathbb{N}$ which computes $f(a, y)$ for any given a

Theorem 7.2 *Let $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ be computable. Then there exists a total computable function $h : \mathbb{N} \to \mathbb{N}$ such that $f(x, y) \simeq Q^{(1)}_{h(x)}(y)$.*

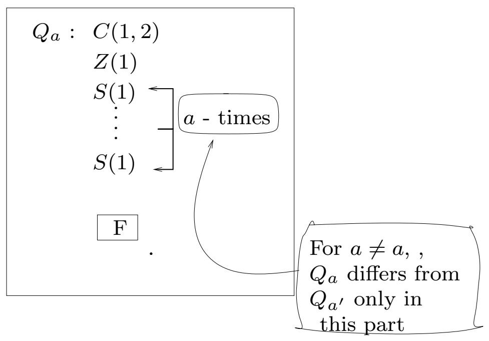
Proof: We need to construct a program Q_a for each a which computes

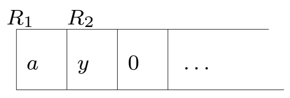
$f(a, y)$ for each y. let F computes f.

□



Initial state of Q_a





Initial state of F

Where F computes f.

Let $k(a) = \ulcorner Q_a \urcorner = \gamma(Q_a)$

Since F is fixed and $\ulcorner\,\urcorner$ is effectively computable, k is effectively computable. Hence $Q_{k_{(a)}}(y) = f(a, y)$

Examples

1. Let $f(x, y) = y^x$. By the $s-m-n$ theorem there is a total computable function $h : \mathbb{N} \to \mathbb{N}$ such that for any fixed $n \in \mathbb{N}$, $h(n)$ is the Godel number of the program that computes y^n.

2. Let $f(x, y) = \begin{cases} y & if \quad x/y. \\ \perp & otherwise. \end{cases}$

   Then f is computable since

   $f(x, y) = x.\Big(\mu z[x.z = y]\Big)$

$$\boxed{\begin{array}{l} x = 0, y = 0 \Rightarrow f(x, y) = 0 = y \\ x = 0, y \neq 0 \Rightarrow f(x, y) = \perp \\ x \neq 0, y = 0 \Rightarrow f(x, y) = 0 = y \\ x \neq 0, y \neq 0 \Rightarrow f(x, y) = \begin{cases} y \leftarrow x/y \\ \perp \leftarrow x \nmid y \end{cases} \end{array}}$$

   Then there is a total computable function $h : \mathbb{N} \to \mathbb{N}$. such that for each fixed n.

   $Q_{h(n)}(y)$ is defined iff n/y iff y is in the range of $Q_{h(n)}$

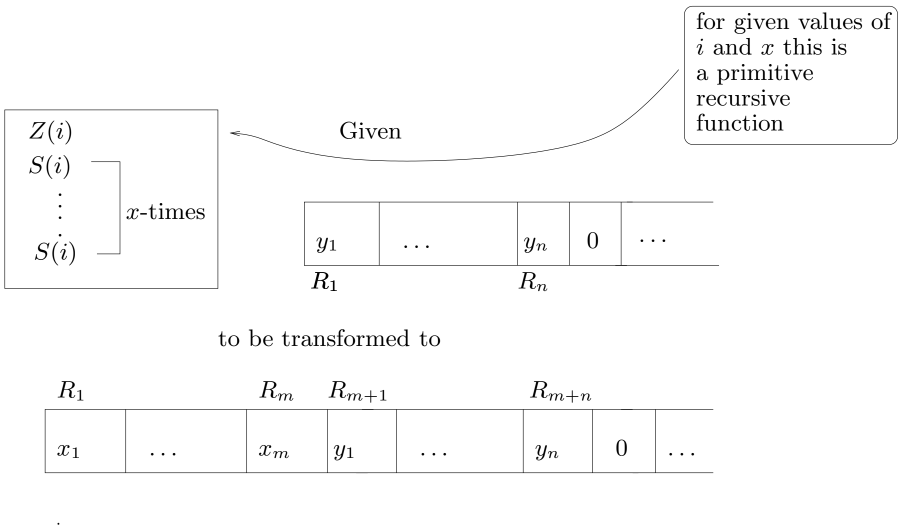
   $therefore$ $\boxed{Dom\Big((Q_{h(n)}(y)\Big) = \{ni | i \in \mathbb{N}\} = n\mathbb{N} = Range\Big(Q_{h(n)}(y)\Big)}$

The generalized S_{mn} theorem

For each $m, n \geq 1$, there is total computable $(m + 1)$-ary function $s^m_n(e, \overrightarrow{x})$ such that

$$\boxed{Q_e^{(m+n)}(\overrightarrow{x}, \overrightarrow{y}) = Q_{s_n^m(e, \overrightarrow{x})}(\overrightarrow{y})} \text{ for all } \overrightarrow{x}, \overrightarrow{y}.$$

Proof: Let $Q(i,x)$ be the subroutine □


for given values of i and x this is a primitive recursive function
Given
$Z(i)$
$S(i)$
$S(i)$
x-times
y_1 … y_n 0 …
R_1
R_n
to be transformed to
R_1 R_m R_{m+1} R_{m+n}
x_1 … x_m y_1 … y_n 0 …

we use the following code

```
C(n, m+n)
   ⋮
C(2, m+2)
C(1, m+1)
Q(1, x_1)
   ⋮
C(m, x_m)
P_e
```

From this definition and the effectiveness of ⌜ ⌝ and ⌜ ⌝$^{-1}$ it follows that $s_n^m(e, \overrightarrow{x})$ is effectively computable.

Since this code contains no jumps, it is primitive recursive

7.2 Universal functions and Programs

Definition 7.1 *The universal function for n-ary computable functions is the (n+1)-ary function*

$$\psi_u^{(n)}(e, x_1, ..., x_n) \simeq Q_e^{(n)}(x_1, ..., x_n)$$

Question. Is $\psi_u^{(n)}$ a computable functions ?

Theorem 7.3 *For each $n \geq 1$, the universal function $\psi_u^{(n)}$ is computable.*

Proof outline: Assume a fixed $n > 0$, given a number e, decode the program P_e and mimic the computation of P_e step by step by writing down the configuration of the registers and the next instruction to be executed. If and when this computation stops, the required value of $Q_e^{(n)}(\overrightarrow{x})$ will be in R_1.

Configuration: The registers are coded by

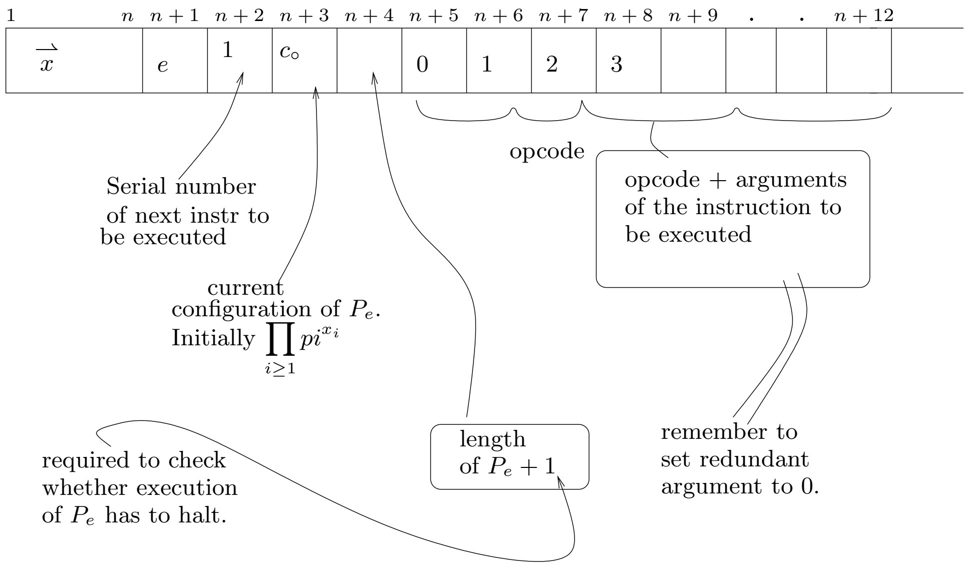
$$c = \prod_{i \leq 1} pi^{ri}$$

where $!R_i = r_i$ and p_i is the $i-th$ prime number. The current state of any computation is given by

$$\sigma = \pi(c, j) \qquad \text{with} \qquad c = \pi_1^{-1}(\rho) \quad j = \pi_2^{-1}(\rho).$$

where j is the serial number of the next instruction to be executed .

Convention. $j = 0$ if the computation is to be stopped.



1
n
n + 1
n + 2
n + 3
n + 4
n + 5
n + 6
n + 7
n + 8
n + 9
.
.
n + 12
x⃗
e
1
c∘
0
1
2
3
opcode
opcode + arguments of the instruction to be executed
Serial number of next instr to be executed
current configuration of Pe. Initially ∏_{i≥1} pi^{xi}
length of Pe + 1
remember to set redundant argument to 0.
required to check whether execution of Pe has to halt.

After decoding the arguments

$$\left.\begin{array}{l} J(n+9, n+5, \quad) \\ J(n+9, n+6, \quad) \end{array}\right\rangle \text{Same target for } Z \text{ and } S \text{ instructions}$$
$$J(n+9, n+7, \quad)$$

$$e = 2^{a_1}(1 + 2^{a_2+1}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))) - 1$$

$$a_1 = \mu z_1 \leq e[2^{z_1+1} \nmid e] \qquad , \qquad \tau_{-1}^{-1}(e) = (e+1)/2^{a_1}, \quad tl(e) = \frac{\tau_{-1}^{-1}(e)}{2} - 1$$

$a_2 = \mu z_2 \leq e[2^{z_2+1} \nmid tl(e)]$ $\qquad$ $hd(e) = \mu z \leq e\,[\,2^{2+1} \nmid \,e\,]$

$a_3 = \mu z_3 \leq e[2^{z_3+1} \nmid tl^2(e)]$ $\qquad$ for $1 \leq i \leq r$

$\vdots$

$a_i = \mu z_i \leq e[2^{z_i+1} \nmid tl^{i-1}(e)]$ $\qquad$ i-th $(e,i) = hd(tl^{i-1}(e))$

therefore $n - th(e,i) = \left\{ \right.$

$\psi_U^{(n)}(e, \overrightarrow{x}) = execute(e, \overrightarrow{x}) = c_1(e, \overrightarrow{x}, \mu t[next(e, \overrightarrow{x}, t) = 0])$

$conf(e, \overrightarrow{x}, t) = \{$ configuration ofter t- step of execution

$finalconf(e, \overrightarrow{x}) = conf(e, \overrightarrow{x}, \mu t[next(e, \overrightarrow{x}, t) = 0])$

Assume given $\overrightarrow{x}, e$ in the registers $R_1, ..., R_{n+1}$

set $R_{n+2} \leftarrow 1$.

$R_{n+3} \leftarrow c(\overrightarrow{x}) = \prod_{i \geq 1} pi^{x_i}$

$R_{n+4} \leftarrow len(Pe)$

$S(n+4)$

$J(n+2, n+4, 0)$ $\qquad$ to the end of the program if P_e has to halt.

$R_{n+5} \leftarrow \underbrace{decode(n+2)}$ — decode the instruction whose serial number is stored in R_{n+2}

β^{-1} — gives various opcodes and registers to

$R_{n+3} \leftarrow execute(n+5)$

while decoding, put the opcode in R_{n+6},
and the arguments in $R_{n+7}, R_{n+8}, R_{n+9}$

$\square$

Lemma 7.4 *Let $\tau : \mathbb{N}^+ \rightarrow \mathbb{N}$ be the coding on non-empty sequences. such that*

$\tau[a_1, ..., a_k] = 2^{a_1}(1 + 2^{a_2+1}(1 + 2^{a_3+1}(...(1 + 2^{a_k+1})...))) - 1 = e.$

Then the following functions are primitive recursive

(i) $hd(e) = a_1$ for the $k \geq 1$

(ii) $tl(e) = \tau[a_2, \ldots, a_k]$ for $k \geq 2$.

(iii) $len(e) = k$. *for* $k \geq 1$

Proof: (i) Consider $e = 2^{a_1}(1+2^{a_2+1}(1+2^{a_3+1}(...(1+2^{a_k+1})...)))-1$. Clearly $e+1$ has at least one non-zero bit in its representation and $e+1 \geq 1$. we have

$$\boxed{hd(e) = \nu z \leq e+1[2^z|(e+1)] = a_1}$$

(ii) Since all sequences are non-empty consider the coding $\tau_\circ : \mathbb{N}^\star \to \mathbb{N}$ defined also for empty sequences with $\tau_\circ[\ ] = 0$ and $\tau \circ [a_k, ..., a_k] \ = \ \tau[a_k, ..., a_k] + 1$.

for all single element sequences $[a]$

$$\tau[a] = 2^a - 1 \qquad \tau_\circ[a] = 2^a$$

$hd[a] = a$. Hence for any $e \geq 0$ if $e+1$ is a power of 2, then is a single element sequence

□

$$ispwr2(x) = \begin{cases} 0 \leftarrow x = 0 \\ \\ \exists\, z \leq x[2^z = x] \end{cases}$$

$len(e) = \mu k \leq (e+1)[ispwrof\ \ 2(tl^k(e+1))]$

for all $e \geq 0$, $e+1 > 0$ and has at least one non-zero bit in its representation.

$$\tau[a_1, ..., a_k] = \begin{cases} 2_{a_1} - 1 \ \text{ if } k = 1 \\ \\ (2\tau[a_2, ..., a_k] + 1)2^{a_1 - 1}) \ \text{ if } k > 1. \end{cases}$$

$$e = (2\, e' + 1)2^{a_1} - 1$$

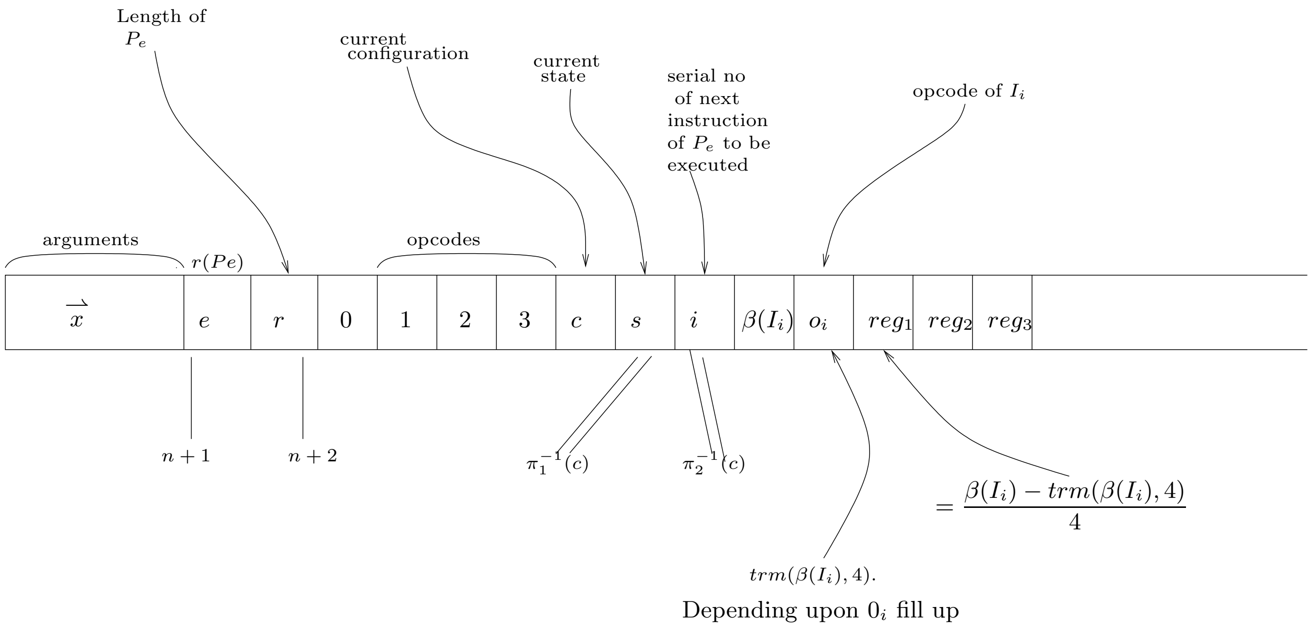
$$therefore\ \frac{e+1}{2^{a_1}} = 2e' - 1$$

$$therefore\ \ e' = \frac{\left(\frac{(e+1)}{2^{a_1}} - 1\right)}{2} = \frac{(e+1) - 2^{a_1}}{2^{a_1+1}} = \frac{(e+1) - 2^{hd(e)}}{2^{hd(e)+1}}$$

$$\boxed{len(e)\mu k \leq e+1\,[tl^k(e) = 0]}$$

$$s = \prod_{j \geq 1} \text{ where } r_j =!R_j$$

$$c = \pi(s, i)$$



$P_e = [I_1, ..., I_r]$

$e = \tau[I_1, ...I_r]$

$= 2^{a_1}(1 + 2^{a_2+1}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))) - 1$

$\tau[0] = 2^0 - 1 = 0$

$e = 2^{a_1}(1 + 2^{a_2+1}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...)))^{-1}$

$hd(e) = a_1$

$tl(e) = 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))) - 1$

$$\frac{e+1}{2^{a_1}} = 1 + 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))$$

$$\frac{e+1}{2^{a_1}} - 1 = 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))$$

$$\frac{(\frac{e+1}{2^{a_1}} - 1)}{2} = 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))$$

$$\frac{e+1}{2^{a_1+1}} - \frac{1}{2} = 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))$$

$$\frac{2(e+1)-1}{2^{a_1+1}} = 2^{a_2}(1 + 2^{a_3+1}(...(1 + 2^{a_r+1})...))$$

$$\frac{2(e+1)-1}{2^{a_1+1}} - 1 = tl(e)$$

Universal Programs : Summary

Assume for any arguments $\overrightarrow{x}$ and Gödel number e, the program P_e is in some configuration $c = \pi(s, j)$ then we may define the function

$\boxed{move(\overrightarrow{x}, e, c) = c'}$ which is primitive recursive

$$\text{where } c' = \begin{cases} \pi(s,0) \text{ if } i > len(e) \text{ or } i = 0 \\ \pi(s', i') \text{ if } i \leq len(e) \end{cases}$$

$$\text{where } i' = \begin{cases} i+1 \text{ if opcode } (i) < 3 \\ i'' \text{ if opcode } (i) = 3 \text{ and } \pi_2^{-1}(i) = i'' \end{cases}$$

$$\text{and } s' = \begin{cases} \frac{s}{p_j^{r_j}} \text{ if opcode } (i) = 0 \text{ and first } (i) = j \\ sp_j \text{ if opcode } (i) = 1 \text{ and first } (i) = j \\ \frac{sp_k^{r_j}}{p_k^{r_k}} \text{ if opcode } (i) = 2 \text{ and first } (i) = j \\ \qquad\qquad \text{and second } (i) = k \\ s \quad \text{if opcode } (i) = 3 \end{cases}$$

where $r_j = \nu j \leq s[p_j^j | s]$

$r_k = \nu k \leq s[p_k^k|s]$

We may the define the function

$$\boxed{\begin{array}{l} \text{Steps } (\overrightarrow{x}, e, c, 0) = c \\ \\ \text{steps } (\overrightarrow{x}, e, c, t+1) = move(\overrightarrow{x}, e, step(\overrightarrow{x}, e, c, t)) \end{array}}$$

and finally we may define for any configuration c,

$state(c) = {\pi_1}^{-1}(c)$

$nextinstr(c) = \pi_2^{-1}(c)$

$output(c) = \nu k \leq c[p_1^k|state(c)]$

and

$$\boxed{\begin{array}{l} \quad execute(\overrightarrow{x}, e) = output(c_f) \\ \text{where } c_f = steps(\overrightarrow{x}, e, c_0, \underbrace{\mu t[nextinstr(steps(\overrightarrow{x}, e, c_0, t)) = 0]}) \end{array}}$$

This function is not prim.rec it is however partial recursive since it is an unbounded minimization of a prim.rec function

Corollary 7.5 *For any URM program P and inputs* $\overrightarrow{x}$ *with* $|\overrightarrow{x}| = n \geq 1$, $Q^{(n)}_{r(p)}$ *is a partial recursive function i.e for any* $n \geq 1$, $URM^{(n)} = PARTREC^{(N)}$

Undecidability : The Hallting Program

Theorem 7.6 *The predicate "programs x halts on input y " defined by halt(x,y)=* $\begin{cases} 1 & \textit{if } Q_x(y)\downarrow \\ 0 & \textit{otherwise.} \end{cases}$

is undecidable i.e. its characteristic function halt(x,y) is not URM-computable.(equivalently partial recursive)

Proof: By contradication. If halt is computable then so is the function $f(x) = halt(x,x)$[1]. and there exists a URM program P_f which computes f. Now consider he program P'_f constructed as follows.let

$$\boxed{\begin{array}{l} P_f{}' :: \quad \boxed{P_f} \\ \quad p+1.\ Z(m+1) \\ \quad p+2.\ S(m+1) \\ \quad p+3.\ J(1, m+1, p+3) \end{array}}$$

$m = \rho(P_f)$ and P_f consist of p instructions what does P'_f computes ? It computes the function.

$$f' = \begin{cases} 0 \text{ if } & f(x) = 0 \\ \perp \text{ if } f(x) = 1 \end{cases}$$

[1]whether program x halts on input x

Let $r(P'_f) = y$. Then $f'(y) = \begin{cases} 0 \text{ if } \quad f(y) = 0 \Leftrightarrow halt(y,y) = 0 \\ \\ \perp \text{ if } \quad f(y) = 1 \Leftrightarrow halt(y,y) = 1 \end{cases}$

$\boxed{\text{PROGRAM } y \text{ HALTS ON INPUT } y \text{ IFF PROGRAM } y \text{ DOES NOT HALT ON INPUT } y.}$ $\square$

Chapter 8

The type-free λ-calculus

In the previous chapters we have studied two models of computability both of which were firmly grounded in the concept of numbers. This was deliberate – a notion of computability which does not involve numbers and operations on numbers would somehow be unsatisfactory. However, throughout our school education in mathematics, numbers have always been *conceptual* rather than *concrete*. In the early years of our education in arithmetic we have dealt mainly with *numerals* i.e. *sequences of symbols* which *represent* numbers. The *interpretation* of these (sequences of) symbols as numbers and operations on numbers leads us to believe that we are computing with numbers. But in reality, the methods we have learned are really methods for manipulating representations rather than concepts themselves. However we have internalised them to such a large extent that we seldom make the distinction between numbers and their representation. The arithmetic we have learnt includes algorithms and methods of computing which utilise the *representation* of numbers instead of numbers themselves. In later years (in secondary school and high school) we gradually progressed to algebra (the use and manipulation of *unknown* or symbolic quantities) and geometry (requiring spatial reasoning invloving points, lines, planes and measures associated with spatial objects). High school mathematics subjects like trigonometry, mensuration, coordinate geometry and the calculus combine spatial reasoning with algebraic reasoning.

Clearly therefore, the question of using and manipulating pure symbols (or pure symbols in conjunction with quantities) is an important aspect of computing and hence needs to be addressed for its computational power.

In this chapter we study two calculi — the λ-*calculus* and *combinatory logic* – that are purely symbol-based and per se free of interpretations. They may be regarded as pure symbol-processing. The notion of a calculus itself recognizes (uninterpreted) symbol manipulation and processing as its main feature. These calculi are important for several reasons.

1. These calculi form the basis for modern functional programming languages in both design and implementation and

2. historically these calculi predate other models of computation.

Syntax. $x, y \in V$, where V is a countably infinite collection of **variables**.

$$\begin{array}{lcll} L & ::= & x & \text{Variable} \\ & | & \lambda x[L] & \lambda\text{-abstraction} \\ & | & (L\ L) & \lambda\text{-application} \end{array}$$

Precedence and Associativity conventions

1. $(L_1\ L_2\ L_3)$ denotes $((L_1\ L_2)\ L_3)$ i.e. application is left associative

2. $\lambda xy[L]$ denotes $\lambda x[\lambda y[L]]$

Free and bound variables

$$V(x) = \{x\}$$

$$V(\lambda x[L]) = V(L) \cup \{x\}$$

$FV(x) = \{x\}$ $\quad FV(\lambda x[L]) = FV(L) \setminus \{x\}$ $\quad V((LM)) = V(L) \cup V(M)$

$FV((L\ M)) = FV(L) \cup FV(M)$ $\quad BV(L) = V(L) - FV(L)$

The meta-operation of substitution(to ensure no "capture of free-variables")

$$\boxed{\begin{array}{l} x\{N/x\} \equiv N \\ y\{N/x\} \equiv y \end{array}} \quad \boxed{(L\ M)\{N/x\} \equiv (L\{N/x\}\ M\{N/x\})}$$

$$\boxed{\begin{array}{ll} (\lambda x[L])\{N/x\} \equiv \lambda x[L] & \\ (\lambda y[L])\{N/x\} \equiv \lambda y[L\{N/x\}] & y \neq x \\ & \& \; y \notin FV(N) \\ (\lambda y[L])\{N/x\} \equiv \lambda z[L\{Z/y\}\{N/x\}] & y \neq x \\ & \& \; y \in FV(N) \\ & \& \; z \notin FV(L) \cup FV(N) \end{array}}$$

α, β, η conversion

$\lambda x[L] \longrightarrow_\alpha \lambda y[L\{y/x\}]$, $y \notin FV(L)$ RENAMING BOUND VARIABLES

$(\lambda x[L]\, M) \longrightarrow_\beta L\{M/x\}$ FUNCTIONAL CALL

REPLACEMENT OF PARAMETERS(FORMAL)

BY ARGUMENTS (ACTUAL)

$\lambda x[(L\, x)] \longrightarrow_\eta L$ EXTENSIONALITY

Closure of conversions under any context

$\longrightarrow_\lambda$ denotes $\longrightarrow_\alpha$, $\longrightarrow_\beta$, $\longrightarrow_\eta$

$$\frac{L \longrightarrow_\lambda L'}{\lambda x[L] \longrightarrow_\lambda \lambda x[L']} \qquad \frac{L \longrightarrow_\lambda L'}{(L\, M) \longrightarrow_\lambda (L'\, M)} \qquad \frac{M \longrightarrow_\lambda M'}{(L\, M) \longrightarrow_\lambda (L\, M')}$$

Question. Verify that the above rules do not lead to capture of free variables.

$\longrightarrow_\lambda^\star = \bigcup_{n\geq 0} \longrightarrow_\lambda^\eta$ where $L \longrightarrow_\lambda^\circ M$ if $L \equiv M$

$L \longrightarrow_\lambda^{k+1} M$ if $\exists N : L \longrightarrow_\lambda^k N \longrightarrow_\lambda M$

λ-Equality $=_\lambda$

$=_\lambda$ is the least relation such that

$L =_\lambda M$ if $L \rightarrow^\star_\lambda M$

$L =_\lambda M$ if $M =_\lambda L$

$L =_\lambda M$ if $L =_\lambda N$ and $N =_\lambda M$.

Combinatory logic & λ-calculus

Schonfinkel 1924

Turing 1937 post 1936

Godel 1931 presented in 1934.

Church 1933,1936,1941

Kleene 1935,1936.

Rosser 1935

Assume an infinite collection of variables symbols $x, y, z, f, g_1 ...$

$\boxed{\text{Assume two distinguished symbols } \mathbb{S} \text{ and } \mathbb{K}}$

The language of CL(combinator) is
expressions generated by the grammar

$$C ::= \mathbb{S} \,|\mathbb{K}\, \underset{\uparrow}{|}\, (C\, C)\, |x$$

application is left associative

A combinator($C_0 ::= \mathbb{K}|\mathbb{S}|C_0\, C_0$) is
term containing only $\mathbb{S}$ and $\mathbb{K}$ and no other variables.

The language of λ is generated by the grammer

$L ::= x \mid \lambda x[L] \mid (L\ L)$

Λ_0 is the collection
of closed λ-terms are combined

$\Lambda \supseteq \Lambda_0$ is the language.

$\mathbb{S}\,L\,M\,N \longrightarrow_W ((L\,N)(M\,N))$

$\mathbb{K}\,L\,M \longrightarrow_W L$

$\mathbb{S}$ stand for stong composition

$\longrightarrow_W$ stand for week contracion

$\mathbb{K}$ stand for constant

$(\lambda x[L]M) \longrightarrow_\beta \{M/x\}L$

$S \overset{df}{=} \lambda\,x\,y\,z\,[((x\,z)(y\,z))]$

$K \overset{df}{=} \lambda\,x\,y\,[x]$

$\{M/x\}x \equiv M$

$\{M/x\}(L\,N) \equiv (\{M/x\}\,L\,\{M/x\}N)$

$\{M/x\}x \equiv M$

$\{M/x\}y \equiv y$

$\{M/x\}\lambda\,x\,[L] \equiv \lambda\,x\,[L]$

$\{M/x\}\lambda\,y[L] \equiv \lambda\,y\,[\{M/x\}L]$

$\{M/x\}(L\,M) \equiv (\{M/x\}\,L\,\{M/x\}M)$

Example. Consider a new combinator $\mathbb{B}$ (for composition) $\mathbb{B}LMN = (L(MN))$

Define $\mathbb{B} \overset{df}{\equiv} \mathbb{S}(\mathbb{K}\mathbb{S})\mathbb{K}$.

$B\,L\,M\,N$

$\equiv\ \mathbb{S}(\mathbb{K}\,\mathbb{S})\,\mathbb{K}\,L\,M\,N$

$\longrightarrow_W\ \mathbb{K}\,\mathbb{S}\,L(\mathbb{K}\,L)\,M\,N$

$\longrightarrow_W\ \mathbb{S}(\mathbb{K}\,L)\,M\,N$

$\longrightarrow_W\ \mathbb{K}\,L\,N(M\,N)$

$\longrightarrow_W\ L\,(M\,N)$

Example. Define the commutation operator which commutes two arguments

$\mathbb{C}\,X\,Y\,Z\ =\ X\,Z\,Y$

Define $\mathbb{C}\ =\ \mathbb{S}(\mathbb{B}\,\mathbb{B}\,\mathbb{S})(\mathbb{K}\mathbb{K})$

Then $\mathbb{C}\,X\,Y\,Z$

$\equiv\ \mathbb{S}\,(\mathbb{B}\,\mathbb{B}\,\mathbb{S})\,(\mathbb{K}\,\mathbb{K})X\,Y\,Z$

$\longrightarrow_W\ \mathbb{B}\,\mathbb{B}\,\mathbb{S}\,X(\mathbb{K}\,\mathbb{K}\,X)Y\,Z$

$\longrightarrow_W\ \mathbb{B}\,\mathbb{B}\,\mathbb{S}\,X\,\underbrace{\mathbb{K}\,Y\,Z}$[1]

$\longrightarrow_W\ \mathbb{B}\,(\mathbb{S}\,X)\,\mathbb{K}\,Y\,Z$

[1]This is not a relex if you parenthesize it complete

Bibliography

[1] Hopcroft J and Ullman J D. *Automata, Languages and Computation.* Addison-Wesley, New York, 1983.

[2] Davis M. *Computability and Unsolvability.* Dover Publications Inc., New York, 1982.

[3] Davis M and Weyukar E. *Computability, Complexity and Languages.* Academic Press, New York, 1983.

[4] Cutland N. *Computability: An Introduction to Recursive Function Theory.* CUP, Cambridge, Great Britain, 1980.